NASA/CP—1999–209236

SEL–98–002
Software Engineering Laboratory Series

Proceedings of the Twenty-Third Annual Software Engineering Workshop

Compiled by:
Goddard Space Flight Center

Proceedings of a workshop held
at the Goddard Space Flight Center
Greenbelt, Maryland
December 2-3, 1998

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

June 1999

The NASA STI Program Office ... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA's counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or cosponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and mission, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at http://www.sti.nasa.gov/STI-homepage.html
- E-mail your question via the Internet to help@sti.nasa.gov
- Fax your question to the NASA Access Help Desk at (301) 621-0134
- Telephone the NASA Access Help Desk at (301) 621-0390
- Write to:
  NASA Access Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076–1320

The views and findings expressed herein are those of the authors and presenters and do not necessarily represent the views, estimates, or policies of the SEL. All material herein is reprinted as submitted by authors and presenters, who are solely responsible for compliance with any relevant copyright, patent, or other proprietary restrictions.

Available from:

NASA Center for AeroSpace Information
7121 Standard Drive
Hanover, MD 21076–1320
Price Code: A17

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Price Code: A10

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1976 and has three primary organizational members:

NASA/GSFC, Information Systems Center

The University of Maryland, Department of Computer Science

Computer Sciences Corporation, Development and Sustaining Engineering Organization

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effects of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

Documents from the Software Engineering Laboratory Series can be obtained via the SEL homepage at:

http://sel.gsfc.nasa.gov/

or by writing to:

Systems Integration and Engineering Branch
Code 581
Goddard Space Flight Center
Greenbelt, Maryland 20771

CONTENTS

Materials for each session include the viewgraphs presented at the workshop and a supporting paper submitted for inclusion in these *Proceedings.*

CONTENTS (cont'd)

CONTENTS (cont'd)

Session 1: The Software Engineering Laboratory

Goddard's New Integrated Approach to IT
H. Kea, NASA/Goddard

Baselining the New GSFC Information Systems Center: the Foundation for Verifiable Software Process Improvement
A. Parra, D. Schultz, J. Boger, and S. Condon, Computer Sciences Corporation,
V. Basili, R. Webby, M. Morisio, D. Yakimovich, and J. Carver,
University of Maryland,
S. Kraft and J. Lubelczyk, NASA/Goddard

Using Experiments to Build a Body of Knowledge
V. Basili, University of Maryland

GODDARD'S NEW APPROACH TO INFORMATION TECHNOLOGY

The Information Systems Center An Overview

The 23rd Annual Software Engineering Workshop

December 2-3, 1994

By Howard E. Kea

PREFACE
By Martha Szczur

Welcome and Al Diaz Introduction
23rd GSFC Software Engineering Workshop
December 2, 1998

Hi, I'm Marti Szczur, the Chief of the Information Systems Center, which is one of the organizations within the Applied Engineering & Technology Directorate (AETD).

Since last year's workshop, Goddard has undergone a significant reorganization. AETD is one of two new directorates, made up of over 1300 Goddard engineers, including computer science professionals. The engineers are matrixed or assigned to flight projects, science directorate activities and/or advanced technology tasks. ISC is one of the engineering groups within AETD, and as the name implies, the Information System Center is heavily vested in all aspects of software (from design, development, testing, validation, integration, maintenance, and including assessment of existing software products.)

The software is applied to a broad spectrum of mission and science systems ... from command & control of the spacecraft (both on-board and on the ground) to planning/scheduling, guidance & navigation systems, communication support, to the processing, archival, & distribution and analysis of science data ... Software is one of the key business products within the ISC.

And thus, my interest in software engineering is extremely high. In fact, the Software Engineering Lab, the group hosting this workshop, resides within the ISC, and I am a strong supporter of the research they conduct. I'm also interested in their expanding their software engineering knowledge and influence across Goddard, as well as NASA. Because of my vested interest in SE as a computer science discipline, it is quite a privilege for me to be opening this 23rd Software Engineering workshop.

I'd like to mention a recent exercise at Goddard, which involved looking ahead to the year 2003 and defining the type of work and missions in which we would be involved. And, the future missions identified have increasing software complexity, such as

- operation of multiple spacecraft and constellations
- distributed sensing systems
- increased on-board science processing and autonomous operations
- higher volume/higher rate of science data to process, manage, archive and distribute
- collaborative, distributed engineering and science computing environments to improve formulation and implementation of missions, as well as to foster collaborative scientific discovery.

To meet these software challenges, It is critical that advancements in software engineering be made. Today, the software industry has not been overly successful in consistently developing software systems that are within budget or on time or which meet all the requirements.

For example, in a Standish Group's 1994 study*, based on an evaluation of 8330 industry software projects, only 16% were actually successful in being on-time, in budget and meeting all originally-specified requirements,

A staggering 53% were "challenged". On an average, they were (1) 189% over budget, (2) had time overruns of 222% and (3) only 61% of originally specified requirements were met.

The other 31% of the software projects were canceled somewhere during development.

Thus, with the increase of NASA mission's dependency on software and the increase in its' complexity, a focus on producing quality software, and thus software engineering, I feel, becomes a critical necessity.

And, it is many of you in this room who will move us in a direction to enable a time when we can develop software systems which are bug-free, reusable, delivered on schedule and within cost while meeting all requirements...on a consistent basis.

Many of the presentations over the next two days pertain to advances and lessons learned which are directly related to the software engineering challenges we face. I look forward to listening and learning from the diverse collection of international experts represented here today.

I have the privilege this morning to be introducing, Al Diaz, who is the Director of Goddard Space Flight Center.

We are very lucky at GSFC because Al, I believe more than any other Center Director to date, has an appreciation of the critical role software ... and in particular QUALITY software ... plays in the success of Goddard's missions, and he recognizes its increasing role in the future.

So, with pleasure, I welcome Al and thank him for agreeing to take time from his incredibly busy schedule to open the 23rd Software Engineering Workshop.

---

* NOTE: The Standish Group International, Inc. is a market research and advisory firm specializing in mission-critical software and electronic commerce. Information about this study can be found on their web site: http://www.standishgroup.com Go to the option titled "Chaos Report."

BACKGROUND

The Goddard Space Flight Center (GSFC) Strategic Implementation Plan (SIP) was published in January 1997. Since the plan was published several centerwide activities have been initiated. One in particular known as "Project Goddard" is responsible for one of the most significant changes that have occurred in Goddard's history. This was the reorganization of Codes 500 and 700. The reorganization [Reference 1] was the result of much planning that began with an assessment of the external environment and the writing of Goddard's SIP followed by definition of macro level processes from which an organization that could support those processes was derived. In today's environment, performance, cost and schedule are three critical elements to the successful execution of a program. The requirements have become an integral factor throughout the development process making it necessary for close customer involvement. The reorganization was primarily structured to more effectively focus engineering talent into teams drawn from the different disciplines. This would facilitate being able to provide products and services which support mission needs aligned with customer requirements.

INFORMATION SYSTEMS CENTER

The ISC was created as part of the Goddard reorganization and was located within the Applied Engineering and Technology (AET) Directorate. Why create an ISC? The creation of ISC was to (1) focus expertise and leadership in information system development. (2)Promote organizational collaboration, partnerships, and resource sharing. (3)Stimulate design /development of seamless end-to-end flight and ground systems. (4) Enable flexibility to effectively support many simultaneous projects by improved access to critical mass of discipline expertise. (5)Enhance career growth and opportunities including multi-disciplinary opportunities and (6) to improve communications among information system professionals. Figure 1, is an Organizational Chart of Goddard after the reorganization showing AETD and System, Technology, and Advanced Concepts (STAAC) as new organizations.

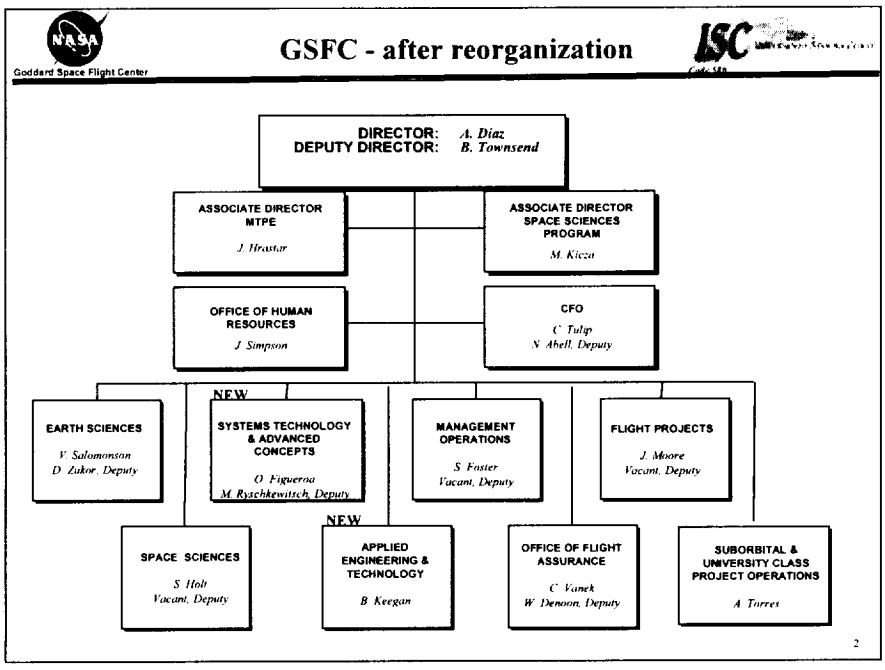


Figure 1.

Figure 2. Shows the AETD Organization, the Director is Brian Keegan.

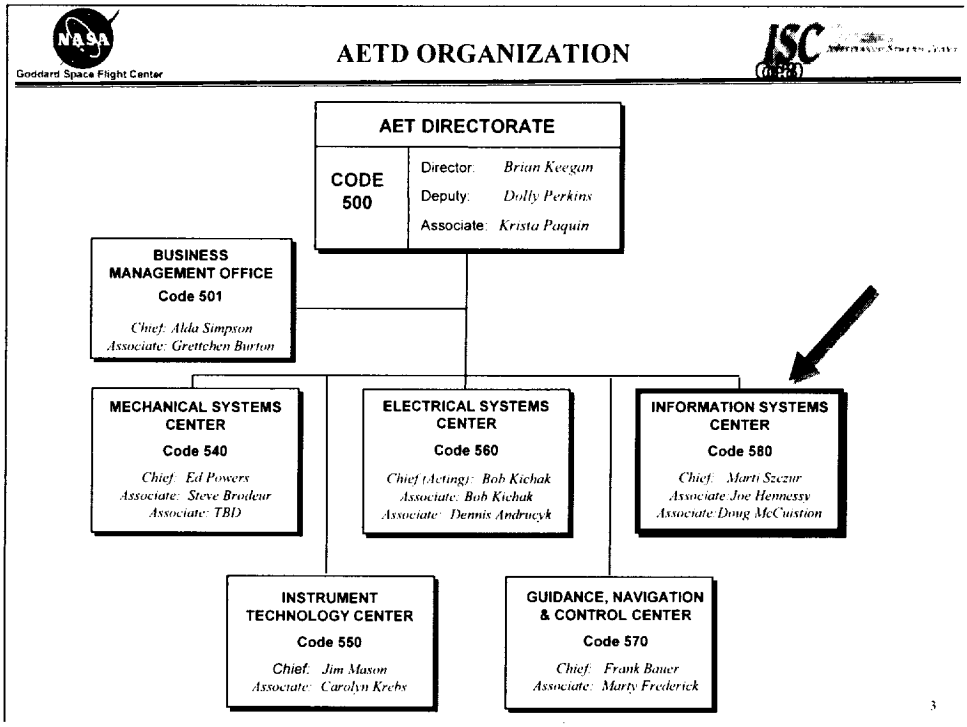


Figure 2.

There are five Engineering Centers within the AETD which are equivalent to Division level organizations. Each of these engineering centers is focused on a particular engineering discipline. The ISC (Code 580) is the engineering center focused on software engineering and computer science. The ISC mission is [Reference 2] "to provide high value information systems products and services and to advance information technologies, which are aligned with customer needs." The ISC organization is shown in Figure 3 below.

ISC has 8 Branches in which each Branch is focused on critical software engineering domains that cover the full lifecycle phase of a mission. Table 1, represents each of the Branches in the ISC and highlights their major functional areas, products and services, customers and projects supported. More detailed information can be found at the ISC Website, http://www.isc.nasa.gov. ISC is predominantly a matrix organization in that many of the Branch personnel 581, 584, 586 are co-located with the project offices. The process in which personnel are assigned is accomplished annually when the projects submit Statements of Work (SOW) to the ISC for services. Personnel with the necessary skills and experience are then assigned to the project from 1 to several years dependent on the duration of the project.

580 / Information Systems Center Branch Structure

Branch	Functional Area/Products	Services	Customer Projects/Org
581 / Systems Integration and Engineering *Leslye Bowce, Howard Kea, Margaret Caulfield*	End-to-end data systems engineering of ISC mission systems development activities.	Mission directors, ground sys/flight ops management, sys. eng., flight prep support, SW eng, Sys I&T, AO prep	EOSDIS, HST, STAAC, NGST, MAP, IMAGE, TRACE, POES, AGS, on-orbit missions
582 / Flight Software *Elaine Shell, Ray Whitley, Lisa Shears*	Embedded spacecraft, instrument and hardware component softwares; FSW testbeds	End-to-end FSW development; simulation s/w; spacecraft sustaining engineering	HST, MAP, TRMM, EO-1, SMEX, SMEX-lite, SPARTAN, EOS AM/PM/Chem, GLAS, XRS XDS, POES, NGST, XTE, EUVE, GRO
583 / Mission Applications *Henry Murray, Scott Green*	Off-line mission data systems (e.g., Command man., s/c mission and science P&S, GN&C, NCC	Sys. eng.& implementation, COTs application, testbeds for concept proof/prototyping in ops environment	NCC SPSR, LS7, EO-1, EOS AM1, HST, TRACE, C930, IMAGE SOC
584 / Realtime Software Engineering *Barb Pfarr, Jay Puttman, John Donohue*	Real-time ground mission data systems for I&T and on-orbit ops (e.g., s/c command & control, launch and tracking services)	Sys. eng.& implementation, COTs application, simulators, testbeds for concept proof/prototyping in ops env.	HST, WFF, ISTP, IMAGE, MAP, SMEX, TRACE,WIRE, EO-1, LS7, HITCHHIKER, SPARTAN, EOS, NGST
585 / Computing Environments and Technology *Howard Eiserike, Steve Naus*	Tools and services in support of information management	Hands-on sys admin., network manage., business/support tool develop, WWW application	EOSDIS, IFMP, C630, C930, HST, WSC, C250, C450, HST
586 / Science Data Systems *Mary Ann Esfandiari, Mary Reph*	Science data systems including data processing, archival, distribution, analysis & info man.	Sys. eng.& implementation, COTs application & integration, testbeds, prototyping	EOSDIS, LS7, TRACE, TRMM, HST
587 / Advanced Data Management and Analysis *M. Esfandiari (Acting), Jim Byrnes*	Advanced concept development for archival, retrieval, display, dissemination of science data	Next-gen req. development, testbed for sys evaluation, prototype products	FAST, NEAR, WIND, ULYSSES, C632, C686, C694, C930, C922
588,/Advanced Architectures & Autonomy *Doug McCuistion (Acting), Julie Breed*	Technology R&D focused on space-ground automation sys. and advanced architectures	Sys. eng & implementation, human-computer eng., technology evaluations, concept prototypes, sw eng. methods	NCC, STAAC, SOMO, Code SM, EOSDIS, MIDEX, NGST

5

Table 1.

The ISC has 4 simple but very critical Strategic Goals to achieve in the next 5 years:

1. Advance leading-edge information systems technology.
2. Clearly define the scope of ISC business, and deliver high value products and services that satisfy customer needs.
3. Build a diverse, talented, innovative, energized, internationally recognized, workforce of employees and managers.
4. Establish open, flexible, collaborative relationships with customers and partners.

These strategic goals are aligned with the Goddard Strategic Goals.

Role of the Software Engineering Laboratory in ISC

Given the external drivers such as "Agenda for Change " which promulgated the creation of the ISC, the SEL has an opportunity to leverage its capabilities to help the ISC meet its strategic goals and objectives. There are several areas where the SEL can be an enabler for software process improvement:[Reference 3]:

- Build an improvement organization within the ISC that will increase the competency of its software engineering professionals, thereby increasing the quality of Goddard software systems.
- Model and characterize software systems in use on the ground and onboard spacecraft.
- Transfer and help tailor proven development and maintenance technologies to new domains, internal and external to GSFC.

As a result of Goddard's organizational changes, a new vision and mission statement and new goals and objectives have been established for the SEL. Over the past several months a series of workshops had been

conducted with the SEL Director's to outline and define the new direction for the SEL and still maintain its heritage over the past 20 plus years. The SEL's new Vision and Mission statement shown in Figure 3, emphasizes continuous software process improvement.

Software Engineering Laboratory Vision:

To be internationally recognized as a leader for applied research in Evolutionary Software Engineering Process Improvement.

Software Engineering Laboratory Mission:

"Serve as a World Class Laboratory dedicated to evolutionary software engineering process improvement and serve as a clearinghouse within GSFC for software engineering best practices. And to foster the development of highly skilled software engineers in the ISC and in GSFC and contractor community through continued education and training of software development practices and methodologies."

Mission Objectives:

1) **To study, research and roll out products from our best practices and methodologies.**
2) **To provide useable and applicable products aligned with customer needs.**
3) **To increase visibility, size and scope.**
4) **To partner with other software engineering organizations.**
5) **To serve as clearinghouse within ISC/GSFC for Software Engineering process improvement information.**
6) **To educate the software engineering community on software engineering best practices.**
7) **To identify resources for funds.**
8) **To develop quickie products e.g. "reusable abstractions" and modularize SEL documents into a handbook format.**
9) **To develop strategies for rolling out practices to customers and immersing customers in the process.**

Figure 3

The current base of SEL activities include: management of databases and producing monthly reports, development of WEB based forms to eliminate file transfer, maintenance of SEL Library and development of Software Engineering Courses. Current research topics include Meta-process, Baseline Process and Core Metrics development. Short term and long term goals for the SEL have been established. They are:

SEL Short-term Goals:

1) Software Engineering Workshop
2) Complete ISC baseline study
3) Update SEL webpage
4) Develop customer focus teams
5) Increase GSFC visibility and interaction

SEL Long-term Goals:

1) Develop a full Software engineering training development program
2) Assist the ISC in obtaining CMM level 2 & 3
3) Establish partnerships with other software Engineering process improvement organizations

Figure 4 shows the relationship of the SEL with ISC. Under the new SEL structure, the ISC Branches and Teams would work more closely with the SEL in defining current processes and developing improved processes. The SEL analysts' role would expand to encompass end-to-end systems development processes, from requirements definition through maintenance and operations. In addition, new metrics will be developed that include the complete lifecycle of the end-to-end systems development process. An example of software technology products supporting the end-to-end mission system is shown in Figure 5.

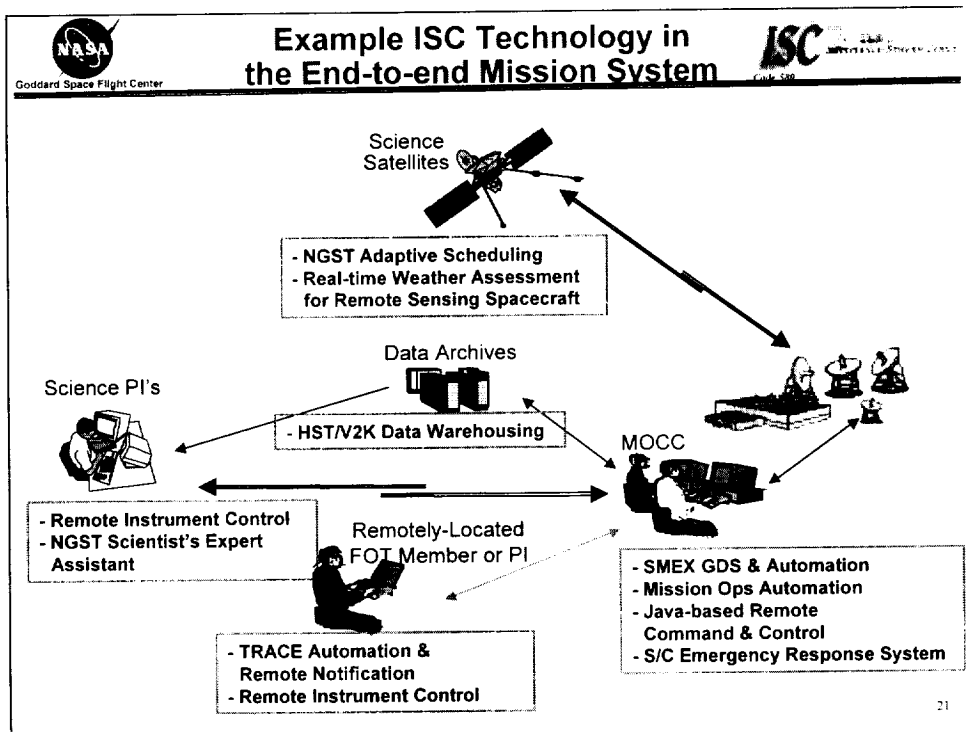


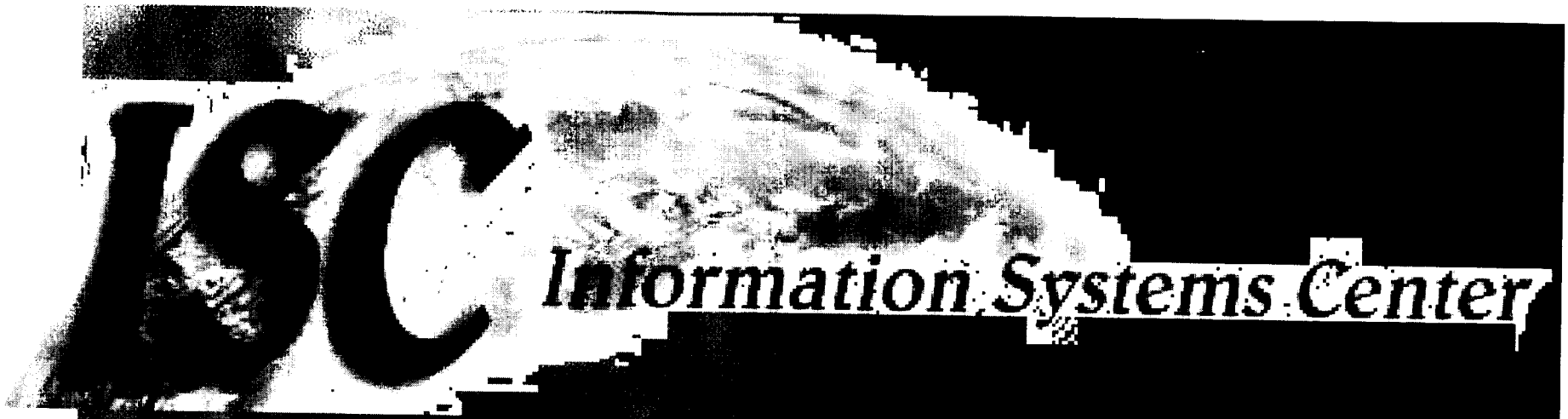
Figure 5.

As a result of the expanded responsibilities, the SEL has already begun to baseline the ISC Branch's products and services and software development processes and team products. This effort will establish a basis for measuring the impact of software process improvement measures that are implemented within the ISC. SEL is also in the process of developing a series of lectures and courses that focus on the Software Engineering Process incorporating the CMM philosophy. The SEL will also play a key role in helping the ISC to achieve CMM levels 2 & 3 and the presence of the SEL in ISC also provides the potential to ultimately achieve CMM levels 4 &5.

In summary, the 23 year history of the SEL has proven that long term focus on continuous improvement can reduce costs and produce a better product. The SEL, as a research organization must continuously adopt to the changing environment in which it exists. Expanding the scope and support activities of the SEL will present a great challenge, however, it will position the ISC to be able to improve Goddard's future systems development efforts.

References:

(1) Keegan, B. "Applied Engineering & Technology Directorate (AETD) 500," AETD Newsletter, NASA Goddard Space Flight Center, August 1998.
(2) ISC Management Team, "ISC Retreat Report", St. Michaels, MD, March 1998.
(3) Pajerski, R. and V. Basili, "The SEL Adapts to Meet Changing Times," Proceedings of the 22nd Annual Software Engineering Workshop, Greenbelt, MD, December 1997.
(4) Szczur, M., "Information Systems Center (ISC) Overview Briefing", NASA Goddard Space Flight Center, May 1998.
(5) Kea, H., "Software Engineering Laboratory Overview," NASA Goddard Space Flight Center, September 1998



Goddard's New Approach to IT

The Information Systems Center
An Overview

The 23rd Annual Software Engineering Workshop

Dec. 2-3, 1998

19-15

ISC Information Systems Center

BACKGROUND

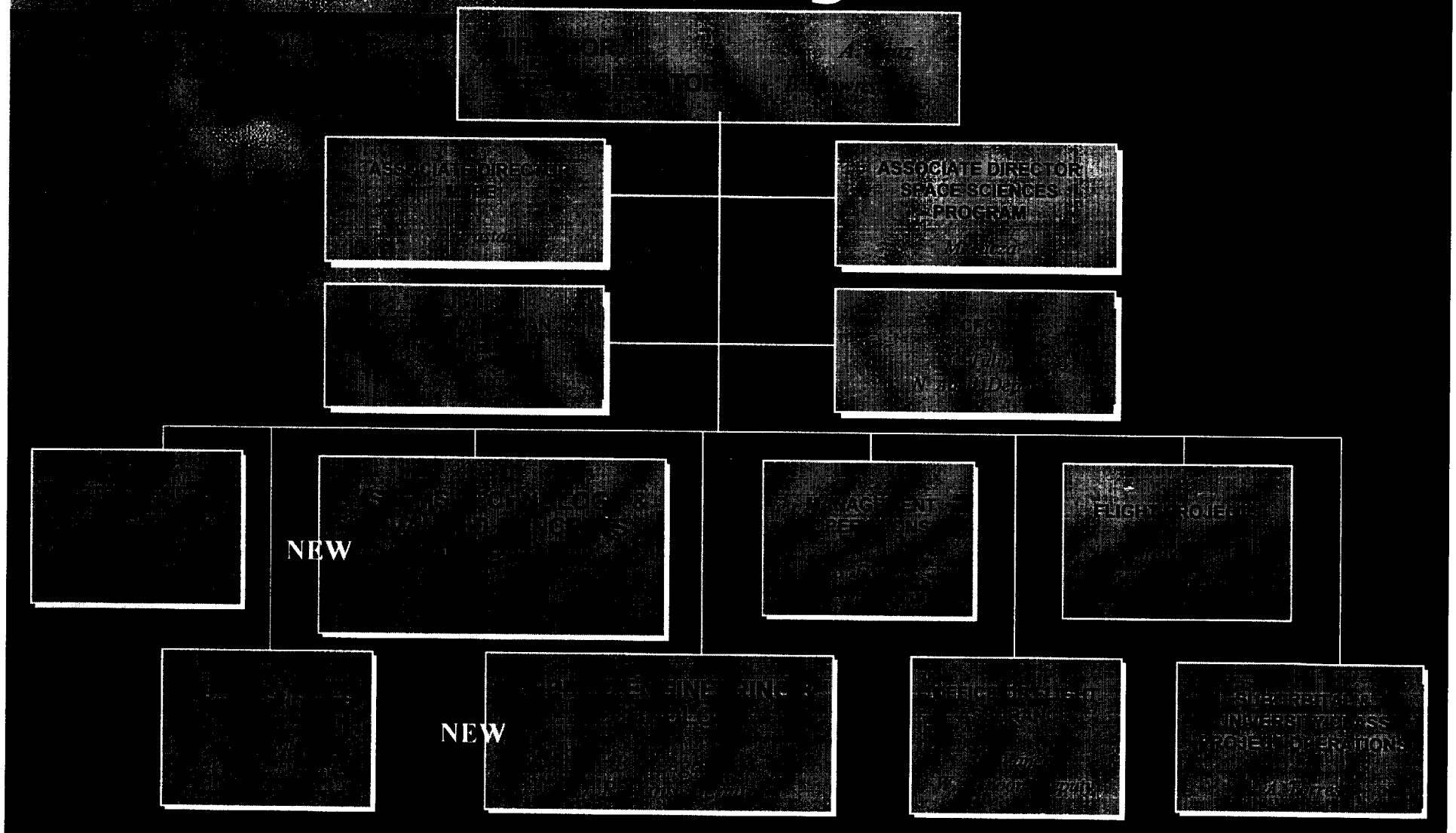
The Goddard Space Flight Center (GSFC) Strategic Implementation Plan (SIP) was published in January 1997.

Several centerwide activities have been initiated such as "Project Goddard".

Genesis for the reorganization of Codes 500 and 700

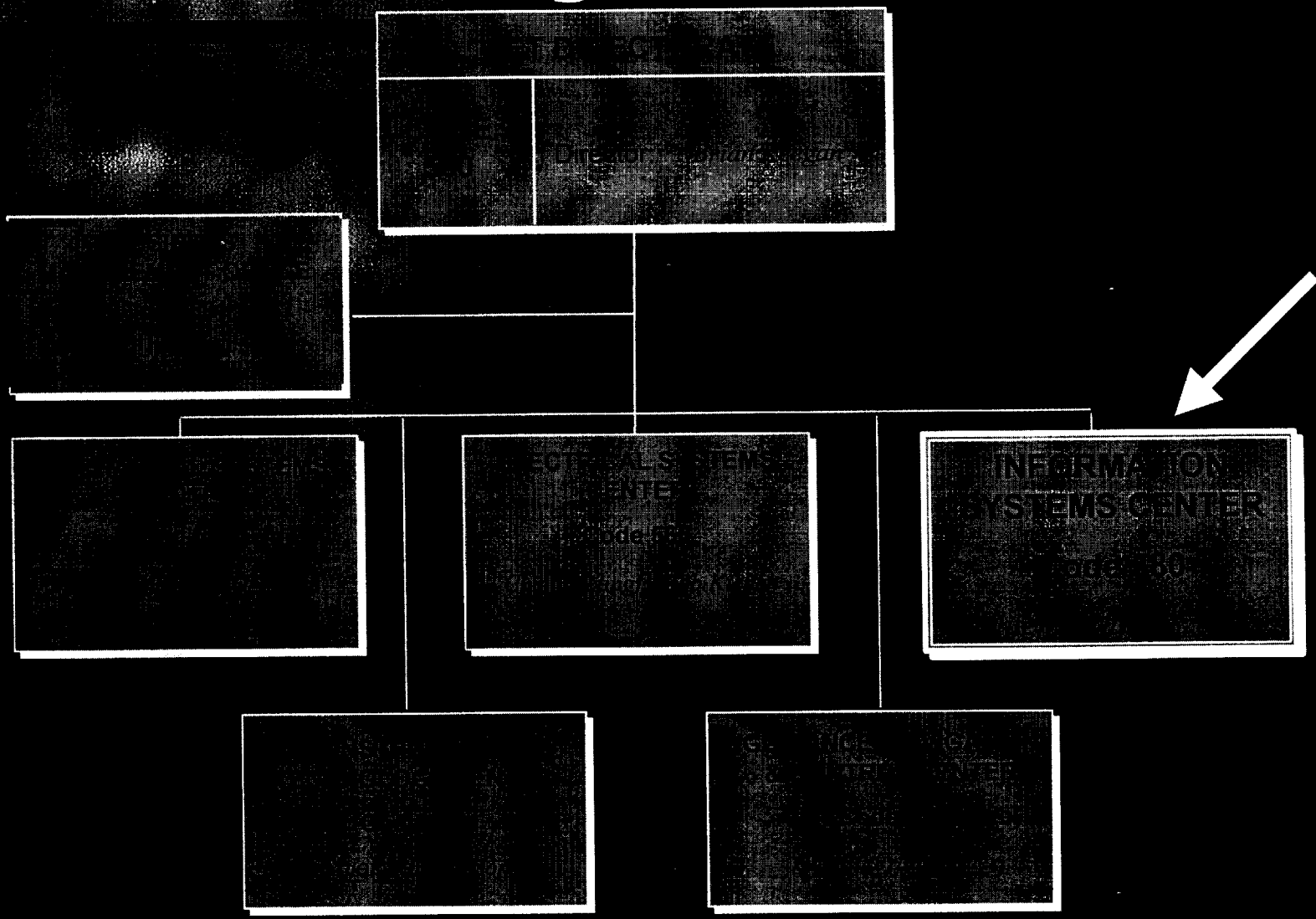
ISC Information Systems Center

GSFC - After Reorganization

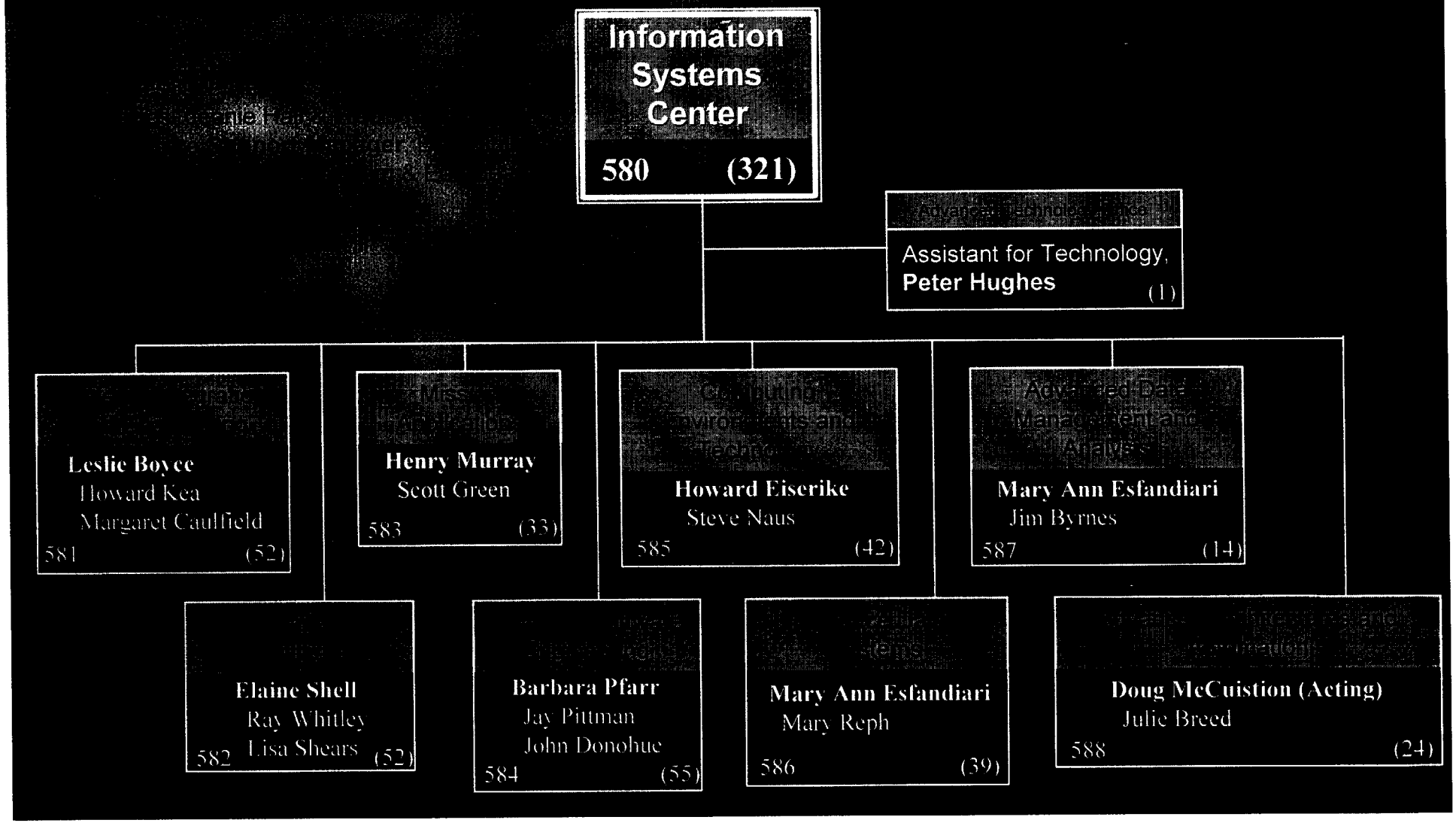


ISC Information Systems Center

AETD Organization



Information Systems Center



580 / Information Systems Center Branch Structure

Branch	Functional Area/Products	Services	Customer Projects/Org
581 / Systems Integration and Engineering *Leslye Boyce, Howard Kea, Margaret Caulfield*	End-to-end data systems engineering of ISC mission systems development activities.	Mission directors, ground sys/flight ops management, sys. eng., flight prep support, SW eng, Sys I&T, AO prep	EOSDIS, HST, STAAC, NGST, MAP, IMAGE, TRACE, POES, AGS, on-orbit missions
582 / Flight Software *Elaine Shell, Ray Whitley, Lisa Shears*	Embedded spacecraft, instrument and hardware component softwares; FSW testbeds	End-to-end FSW development; simulation s/w; spacecraft sustaining engineering	HST, MAP, TRMM, EO-1, SMEX, SMEX-lite, SPARTAN, EOS AM PM Chem, GLAS, XRS XDS, POES, NGST, XTE, EUVE, GRO
583 / Mission Applications *Henry Murray, Scott Green*	Off-line mission data systems (e.g., Command man., s/c mission and science P&S, GN&C, NCC	Sys. eng.& implementation, COTs application, testbeds for concept proof/prototyping in ops environment	NCC SPSR, LS7, EO-1, EOS AM1, HST, TRACE, C930, IMAGE SOC
584 / Realtime Software Engineering *Barb Pfarr, Jay Pittman, John Donohue*	Real-time ground mission data systems for I&T and on-orbit ops (e.g., s/c command & control, launch and tracking services)	Sys. eng.& implementation, COTs application, simulators, testbeds for concept proof/prototyping in ops env.	HST, WFF, ISTP, IMAGE, MAP, SMEX, TRACE,WIRE, EO-1, LS7, HITCHHIKER, SPARTAN, EOS, NGST
585 / Computing Environments and Technology *Howard Eiserike, Steve Naus*	Tools and services in support of information management	Hands-on sys admin., network manage., business/support tool develop, WWW application	EOSDIS, IFMP, C630, C930, HST, WSC, C250, C450, HST
586 / Science Data Systems *Mary Ann Esfandiari, Mary Reph*	Science data systems including data processing, archival, distribution, analysis & info man.	Sys. eng.& implementation, COTs application & integration, testbeds, prototyping	EOSDIS, LS7, TRACE, TRMM, HST
587 / Advanced Data Management and Analysis *M. Esfandiari (Acting), Jim Byrnes*	Advanced concept development for archival, retrieval, display, dissemination of science data	Next-gen req. development, testbed for sys evaluation, prototype products	FAST, NEAR, WIND, ULYSSES, C632, C686, C694, C930, C922
588 / Advanced Architectures & Autonomy *Doug McCuistion (Acting), Julie Breed*	Technology R&D focused on space-ground automation sys. and advanced architectures	Sys. eng & implementation, human-computer eng., technology evaluations, concept prototypes, sw eng. methods	NCC, STAAC, SOMO, Code SM, EOSDIS, MIDEX, NGST

ISC Vision and Mission

The Vision of ISC is to be a world-class information systems center of excellence serving the needs of GFSC and NASA customers.

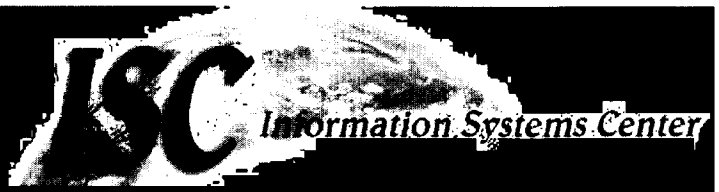
- **ISC has a diverse, talented innovative, energized, internationally recognized, workforce of employees and managers.**
- **ISC is the employer of choice providing a flexible, learning work environment; fair and credible promotions, training, development, and awards; and premier development tools and facilities.**
- **ISC is the leader and focal point for cutting-edge information technology for Goddard's customers in Earth and Space Science and for advanced information technologies to support institutional customers.**
- **ISC delivers innovative, customer-oriented solutions, products and services.**
- **ISC's relationships with customers are open, flexible, collaborative and based on trust and mutual respect.**
- **ISC operates like a business with responsive, efficient, value-added processes, enabling technology infusion/transfer and effective delivery of products and services.**

ISC Information Systems Center

The ISC Strategic Goals

The ISC has 4 simple but very critical Strategic Goals to achieve in the next 5 years they are to:

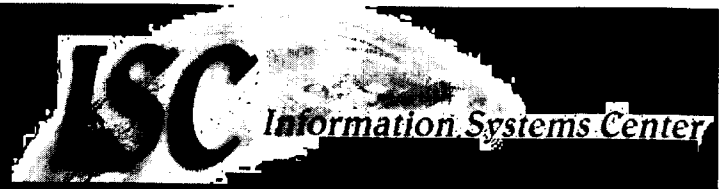
- Advance leading-edge information systems technology.
- Clearly define the scope of ISC business, and deliver high value products and services that satisfy customer needs.
- Build a diverse, talented, innovative, energized, internationally recognized, workforce of employees and managers.
- Establish open, flexible, collaborative relationships with customers and partners



The Software Engineering Lab

The SEL was created in 1976 for the purpose of understanding and improving the overall software process and products. A partnership was formed between NASA/GSFC, the University of Maryland (UM), and Computer Sciences Corporation (CSC), with each of the organizations playing a key role:

- NASA/GSFC/FDD as the user and manager of all of the relevant software systems,
- UM as the focus of advanced concepts in software process and experimentation, and
- CSC as the major contractor responsible for building and maintaining the software used to support the NASA missions.



New SEL Vision and Mission

Software Engineering Laboratory Vision: To be internationally recognized as a leader for applied research in Evolutionary Software Engineering Process Improvement.

Software Engineering Laboratory Mission: "Serve as a World Class Laboratory dedicated to evolutionary software engineering process improvement and serve as a clearinghouse within GSFC for software engineering best practices. And to foster the development of highly skilled software engineers in the ISC and in GSFC and contractor community through continued education and training of software development practices and methodologies."

ISC Information Systems Center

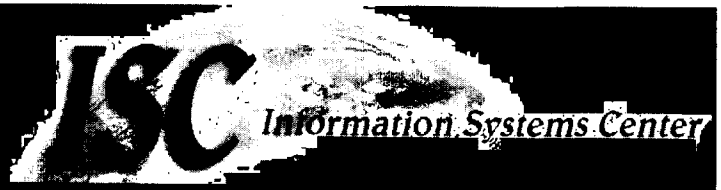
Current SEL Activities

The current base of SEL activities include:

- management of databases and producing monthly reports
- development of WEB based forms to eliminate file transfer
- maintenance of SEL Library
- development of Software Engineering Courses.

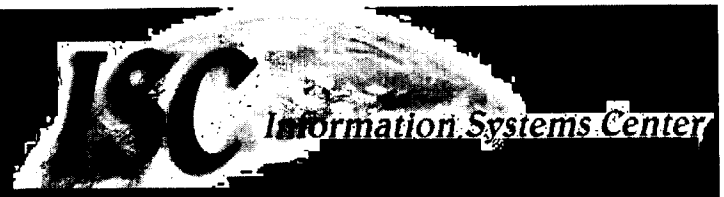
Current research topics include

- Meta-process, Baseline Process and Core Metrics development.



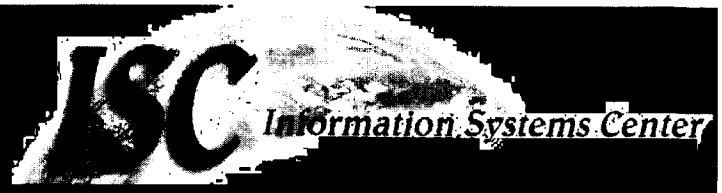
SEL Short-term Goals

- Software Engineering Workshop
- Complete ISC Baseline
- Update Webpage
- Develop customer focus teams
- Develop Short 1 day courses
- Increase GSFC Visibility and Interaction
- Support ISC Reuse Program in Collaboration with Ames IV&V Center



SEL Long-term Goals

- Develop a full Software Engineering Training Development Program
- Investigate benefits for ISC to pursue CMM levels 2 & 3
- Establish Partnerships with other Software Engineering Process Improvement organizations



Role of the SEL in ISC

Build an improvement organization within the ISC that will increase the competency of its software engineering professionals, thereby increasing the quality of Goddard software systems.

Model and characterize software systems in use on the ground and onboard spacecraft.

Transfer and help tailor proven development and maintenance technologies to new domains, internal and external to GSFC.

ISC Information Systems Center

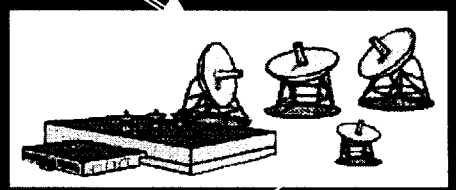
ISC's End-to-end Mission Role

Science Satellites

- NGST Adaptive Scheduling
- Real-time Weather Assessment for Remote Sensing Spacecraft

Data Archives

- HST/V2K Data Warehousing



Science PI's

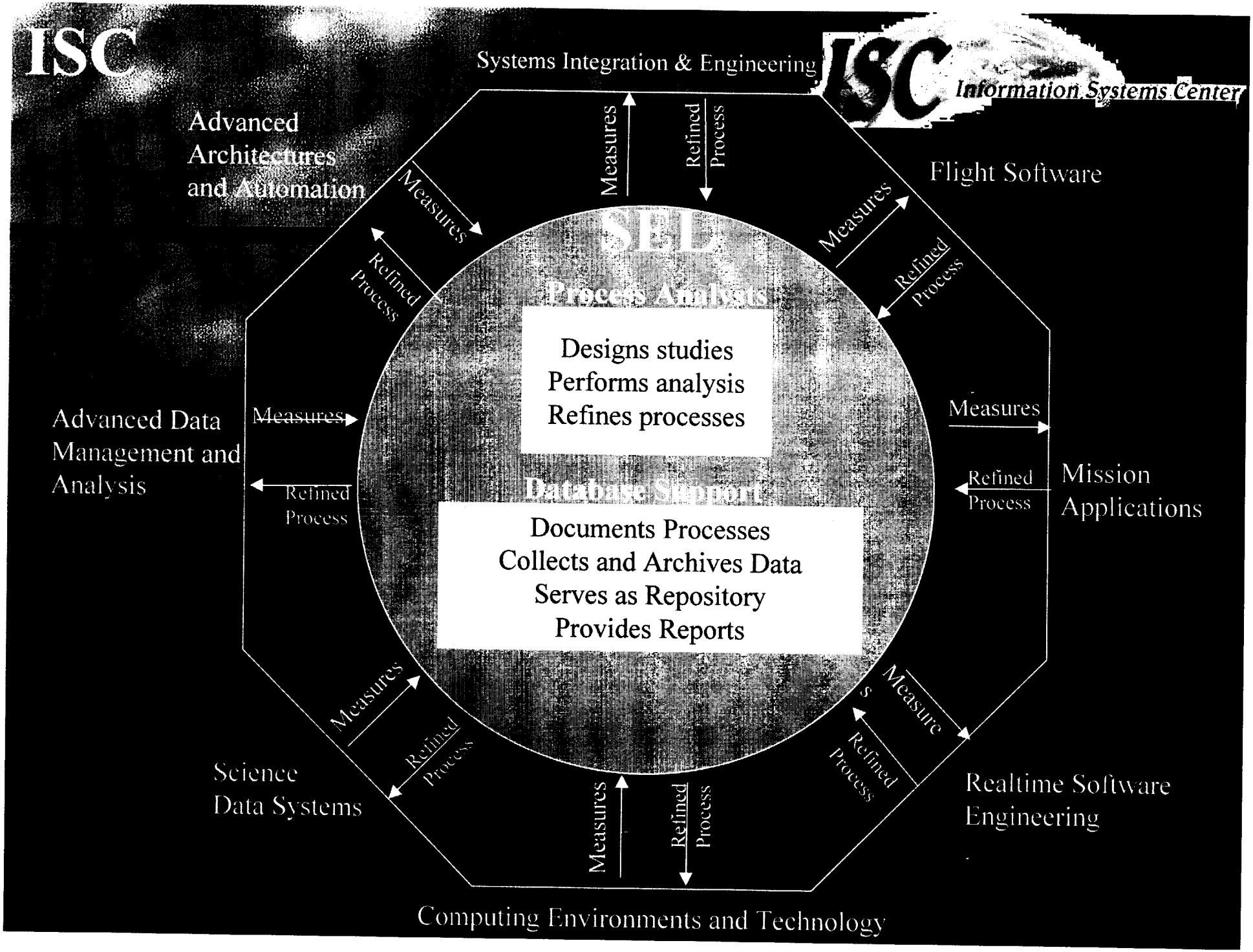
MOCC

- Remote Instrument Control
- NGST Scientist's Expert Assistant

Remotely-Located FOT Member or PI

- SMEX GDS & Automation
- Mission Ops Automation
- Java-based Remote Command & Control
- S/C Emergency Response System

- TRACE Automation & Remote Notification
- Remote Instrument Control


ISC
Systems Integration & Engineering
ISC
Information Systems Center
Advanced Architectures and Automation
Flight Software
Advanced Data Management and Analysis
Mission Applications
Science Data Systems
Realtime Software Engineering
Computing Environments and Technology
Measures
Refined Process
SEL
Process Analysts
Designs studies
Performs analysis
Refines processes
Database Support
Documents Processes
Collects and Archives Data
Serves as Repository
Provides Reports

Baselining the New GSFC Information Systems Center: The Foundation for Verifiable Software Process Improvement

A. Parra, D. Schultz, J. Boger, S. Condon,
CSC
R. Webby, M. Morisio, D. Yakimovich,
J. Carver, M. Stark,
University of Maryland
V. Basili,
Fraunhofer Center Maryland and University of Maryland
S. Kraft,
NASA/GSFC

Abstract

This paper describes a study performed at the Information System Center (ISC) in NASA Goddard Space Flight Center. The ISC was set up in 1998 as a core competence center in information technology. The study aims at characterizing people, processes and products of the new center, to provide a basis for proposing improvement actions and comparing the center before and after these actions have been performed. The paper presents the ISC, goals and methods of the study, results and suggestions for improvement, through the branch-level portion of this baselining effort.

Introduction

At the beginning of 1998, a major reorganization of software engineering functions took place within the NASA Goddard Space Flight Center. A new "Information Systems Center" (ISC) was created with the objective of concentrating and consolidating Goddard's Information Technology (IT) capabilities into one organizational unit.

Within the aegis of this new organization, sits the Software Engineering Laboratory (SEL) [1,7], a twenty-three years old consortium of process and product improvement specialists from three organizations: NASA Goddard itself, the University of Maryland and Computer Sciences Corporation. The SEL had previously focused most of its efforts within the Flight Dynamics Division (FDD), performing process and product improvement studies and software engineering experiments. With the reorganization of software activities at Goddard, its scope now expands to the entire ISC. Therefore there was a need to better understand the wider context that the SEL now found itself within.

Consequently, a "baseline" study was initiated by the SEL in April 1998. The aim of the baseline was to characterize or profile the ISC in terms of its people, processes and products. Each branch and many teams within the ISC were studied for the purpose of completing an *initial* baseline study. We emphasize the word "initial" to indicate that this study is not a detailed baseline in the sense of capturing extensive focussed data about one aspect of the ISC's operations. Rather it is a baseline that will provide an overall high-level profile of the new organization.

Many previous baselines have been conducted within the FDD, as well as at the level of Goddard Code 500 [4], Goddard as a whole [5] and NASA as a whole [6]. The questionnaires developed by the baselining team were heavily based on these earlier studies to enable comparison. Where practical, this paper will compare data from ISC with earlier studies.

This paper documents preliminary data and observations that the SEL has made in baselining the ISC. The ultimate goals of the baselining study are to identify areas for process and product improvement of benefit to Goddard, as well as interesting and novel research areas to pursue. This paper will begin by elaborating upon the goals of the study. It will continue by describing the methods adopted (and their constraints), the data collected, and the preliminary results of the work. The paper concludes with some recommendations for ISC and suggestions for future work for the SEL.

The ISC

Quoting from the ISC home page [8]:

> "The Information Systems Center (ISC) is an innovative center of expertise in the implementation of seamless, end-to-end information systems in support of NASA programs and projects, and specifically the GSFC Earth Science, Space Science and Technology focus areas. The ISC provides leadership and vision in identifying and sponsoring new and emerging information systems technologies."

The ISC is organized in eight branches, each with a unique function. Refer to Figure 1for the organization structure of ISC and Table 1 for the associated products and services. The meaning of boxes line styles will be explained later. The work is organized in various manners: within these branches exist teams that are producing software products and services, there are personnel (and sometimes teams) matrixed to other ISC branches or other Codes at GSFC, and there are cross-branch teams that serve all the ISC with representation from the branches. The detailed organizational structure is explained in [3].

Certain terminology (noted in Italics) is used in this environment and in this paper, especially terminology related to organizational structure. Basic organizational structure is broken down from highest level to lowest, GSFC is divided into 9 *directorates*, including the Applied Engineering and Technology Directorate (AETD), within that there are 5 *Centers*, including the Information Systems Center, within that the eight *branches* mentioned above, within those branches, *teams* of individuals supporting *projects*, such as the Earth Observing System (EOS). Sometimes a person or persons is *matrixed* from one organizational entity to another, so that one group manages the work, while the person(s) maintains their original organizational alliances.

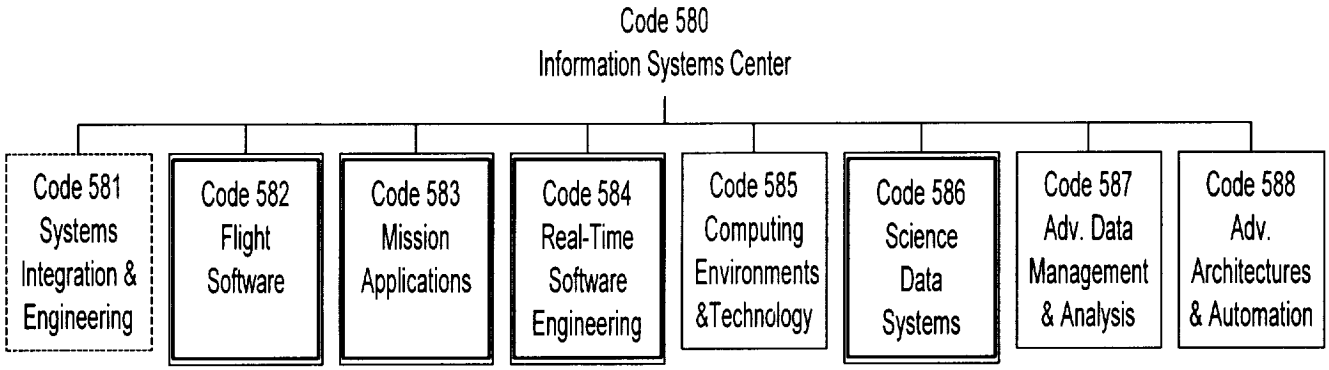


Figure 1 - Organizational Structure of the ISC

Branch Code	***Branch Name***	***Products/Services***
581	Systems Integration and Engineering	End-to-end data systems engineering of ISC mission systems development activities
582	Flight Software	Embedded software products for on-board data handling; management and control of flight hardware
583	Mission Applications	Off-line mission data systems (command management, spacecraft mission planning and scheduling, science planning, etc.)
584	Real-Time Software Engineering	Tools and services in support of information management. Real-time ground mission data systems for I&T and on-orbit ops (e.g., s/c command and control, launch, and tracking services)
585	Computing Environments and Technology	Tools and services in support of information management. Hands-on system administration, network management, WWW applications
586	Science Data Systems	Data processing, archival distribution, analysis and information management for science data systems
587	Advanced Data Management and Analysis	Advanced concept development for archival, retrieval, display, and dissemination of science data
588	Advanced Architectures and Automation	Technology R&D focused on space-ground automation systems and advanced architectures

Table 1. Products and Services of the ISC Branches

Goals for Baselining

The major objective of the baselining study is to gain an understanding of the ISC as to allow us to identify areas for process and product improvement. The philosophy behind the effort is to characterize and understand the new organization before attempting to introduce any new technology or process improvements. From the understanding, we seek to find a basis to assess improvements, which can then be packaged for wider integration into the business. Figure 2 highlights the role of baselining (the bottom rectangle) in the broader context of process and product improvement according to the Experience Factory paradigm [1].

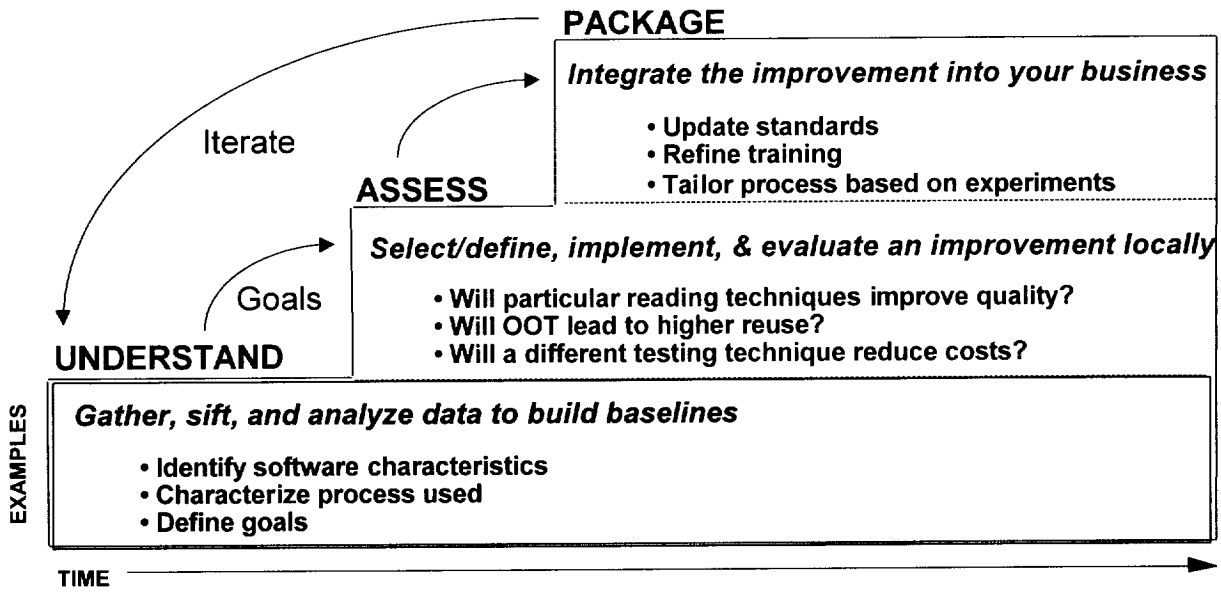


Figure 2 - Role of Baselines in Process and Product Improvement

Methods Used

The following methods, already used in the COTS Study [9], were used.

First, a number of questions and measures have been developed, starting from the high level goals and using the Goal Question Metric (GQM) approach [2], to collect information about ISC's processes, products and people. They gather both quantitative and qualitative information – some of the data are numeric and highly factual (e.g. staff numbers), whereas other data represent informed opinion (e.g., expectations of future change). The aim is to be able to characterize the software products, processes and people within the organization, with adequate qualitative context to meaningfully interpret the hard quantitative data.

Questions and measures have then been organized in a questionnaire and a structured interview [10]. The interview being constrained to no more than 30 – 45 minutes covered the qualitative data. The questionnaire was devoted to quantitative data that were less subject to interpretation.

To enforce consistency, guides for filling questionnaires and performing interviews were developed too [10].

After validating these tools with pilots, they were used to collect data from branch heads and team leaders. The process was the following.

During the interview, the Interviewer asks questions following the outline of the Interview Guide. The Scribe takes notes and employs a tape recorder, if acceptable to the Interviewee, to aid in preparation of the interview report. The Interviewee is told that the result of the interview is the interview report, which will not be considered final until the Interviewee had read and approved it. At the end of the interview the Scribe may ask some clarification questions. The Interviewer gives a copy of the Questionnaire, which asks questions of a detailed, numeric nature that don't lend themselves well to open-ended, face-to-face discussion to the Interviewee, and requests that the Questionnaire be completed within two weeks.

After the interview, the Scribe prepares an interview report, consisting of brief summaries of the Interviewee's responses to the questions on the standard Interview Guide. The Interviewer reviews the notes. Once reviewed they are sent to the Interviewee for concurrence. At this stage of the process, the interview report is considered approved. Tape recordings were not kept as the approved interview report serves as the result of the interview.

At the end of the initial interview, the Interviewer schedules a follow-up interview. The purpose of the follow-up is to go over the questionnaire that the interviewee has completed, and resolve any items where either the questions weren't clear to the interviewee, or the responses are unclear to the interviewer.

About the data

The baseline study collects data at two levels within the ISC: the branch and team levels. The current status of the study is that we have completed the branch data collection and analysis, and are currently finalizing the team-level data collection and the team-level analysis is in progress. Therefore this paper will only report on the results from the branch-level data.

The branch-level data were collected from the management of each branch. Our aim at the branch-level data collection stage was to build an overall characterization of the organization, with a wide range of factors (e.g. process, people, and product) considered. The intent is that we will perform more detailed baselines on specific factors in a subsequent study, as and when more accuracy is required.

The consequence is that the data reported in this paper have varying degrees of reliability. In some cases, they are actual data (e.g. head count). In other cases, they may be derived data. For example, a question asking how much effort was spent on software maintenance versus development was sometimes answered by managers going through their roster and counting how many people did maintenance versus development. In other cases, the data may represent only "guesstimates". Sometimes we asked questions seeking data that they do not collect, so they had to estimate. In all cases, we are dealing with a new organization, so there is not a body of historical data, or even established data collection procedures in many cases.

As we analyze the data, we will report on the expected reliability.

Findings

Domains

Figure 3 presents a depiction of sample application domains in the ISC, in contrast to the more focused domains of the FDD. Whereas the FDD was primarily concerned with attitude, orbit and mission planning applications, the ISC must now be concerned with such diverse pursuits as science data visualization and embedded flight software. The new ISC is a much more heterogeneous organization than the FDD, so the need to understand the context of the data collected is paramount. Direct comparison of branch to branch will be meaningless without an appreciation of the context within which the data were collected.

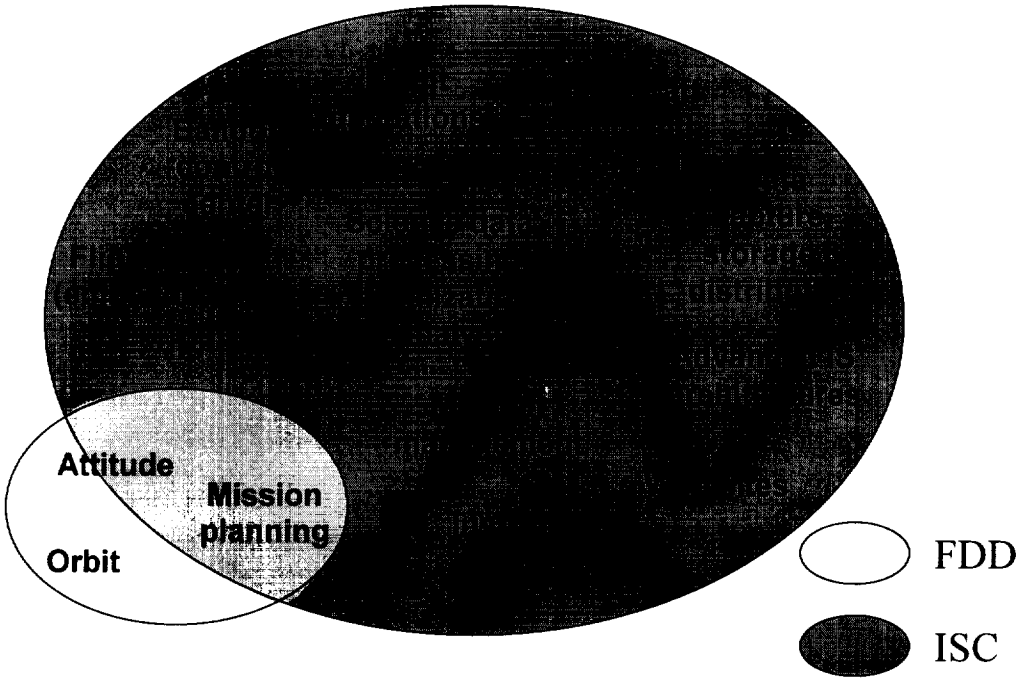


Figure 3. Sample Application Domains in ISC and FDD

Domains and organization

As mentioned above, the Information Systems Center is organizated into eight branches. Figure 1 shows the basic organizational structure of the ISC. We have found that several branches appear to have a functional domain focus (e.g. flight software), specifically these are 582, 583, 584 and 586, designated in Figure 1 with double borders. Those are contrasted with branches that deal primarily with technology domains (e.g. advanced architectures), specifically 585,587 and 588. Code 581 is probably neither in the technology nor functional camp, they deal primarily with the management of systems integration activities, this uniqueness is indicated in Figure 1 with a dashed border.

Matrixing and projects common to branches

In the questionnaire, branch management were asked to list the projects with which their branch was involved. Figure 4 presents the common projects by branch. These are larger projects such as the Hubble Space Telescope (HST) or Landsat-7, where several branches are involved. Another question was the number of staff belonging to the branch but working outside it (or matrixed). On average, 63% of ISC staff is matrixed. Both facts above suggest that the organisation by branches is in some sense virtual, while the projects rather than the branches control the process. This was also confirmed by comments from branch managers. An implication of this for the SEL is that to introduce any process improvement, it would appear necessary to consider how to influence the project to adopt the new technology.

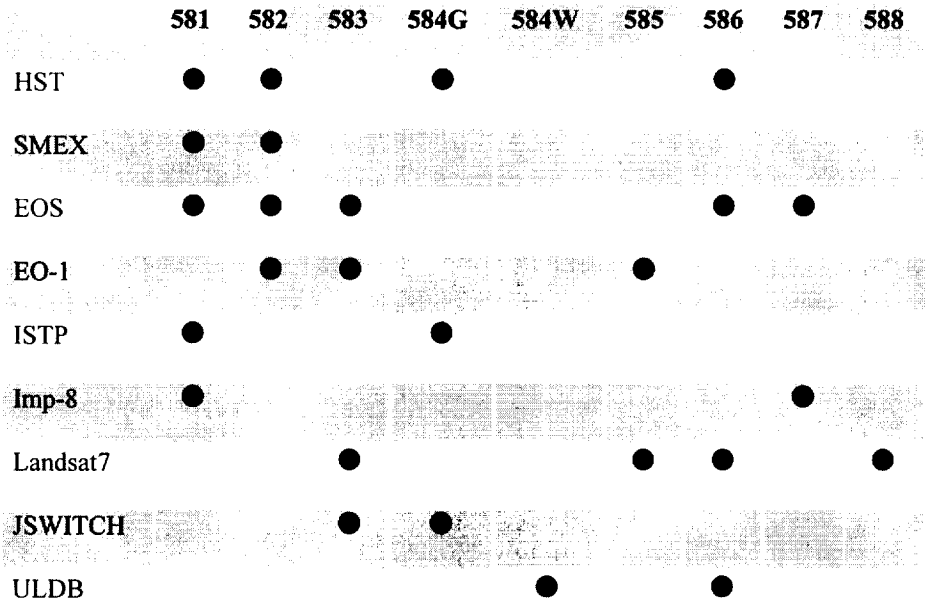


Figure 4 - Common Projects by Branch

Characterization of branches

Figure 5 presents the variation in staff numbers by branch. The total number of civil servants in ISC is 249, based on an aggregation of the questionnaire data. This total has been verified by a check against the overall ISC roster. The total number of contractors in ISC is over 308 – the exact number is difficult to determine because some branches were unable to specify their exact number of contractors[1].

[1]Staffing Numbers - The count of civil servants and subcontractors working for a branch or team is not unique, as they can report to an entity (say the team) but be paid by another (another team or branch or project). Most interviewees did not have both data, and reported the best estimate they had. An effort to collect the most accurate data is underway and will be reported in the ISC Baselining final report.

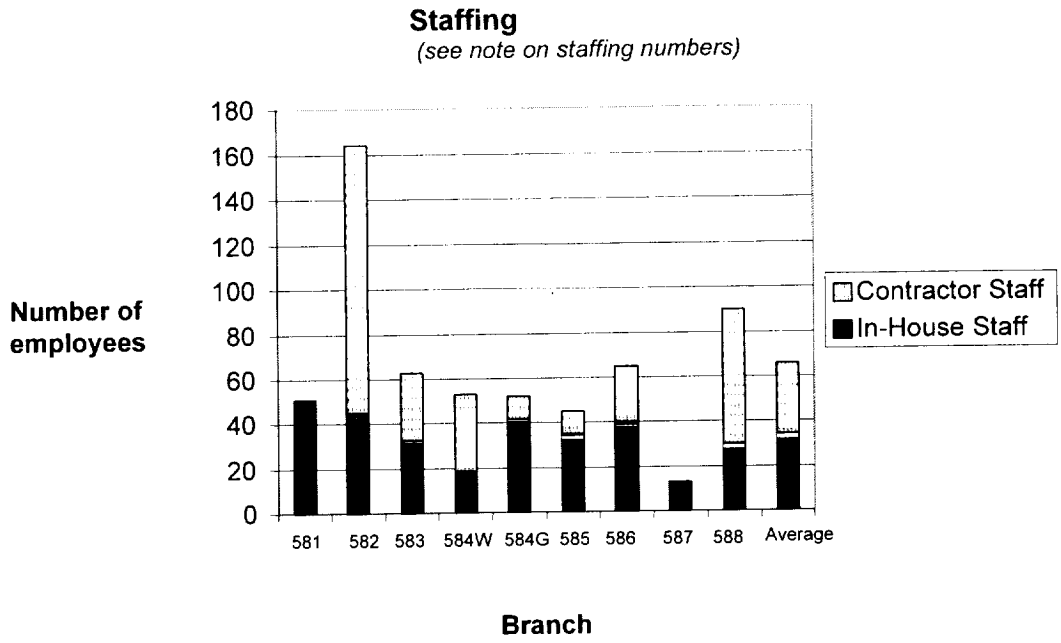


Figure 5 - Staff Numbers by Branch

Most notable here is that there is one very large branch (582), more than 2/3's of its personnel are contractors; one very small branch (587), with no contractors whatsoever; and the rest are mid-sized.

It is worthwhile to compare these figures to the SEL's 1992 baseline of Code 500 [4]. Code 500 at that time contained responsibility for most of the same functional and technology domains that the ISC contains today. Code 500, however, did not employ all of the GSFC software personnel working in these functional and technology domains; the Engineering Directorate (Code 700) employed some of them. On the other side of the balance sheet, however, we must note that some of the 1992 employees of Code 500 were analysts and other "non-software" types. These personnel were largely transferred to "Centers" other than the ISC in the recent GSFC reorganization. With these differences between the Code 500 of 1992 and the ISC of today kept in mind, let us proceed. In the 1992 baseline of code 500, it was found that approximately 1,600 of 5,000 staff (including contractors) were performing software-related functions (development, maintenance, etc). The FDD had 700 staff, of which 250 were in software. This comparison (see Figure 6) indicates that the ISC has approximately twice as many IT-related staff as FDD. However, they are significantly smaller in size than were the code 500 software people in 1992.

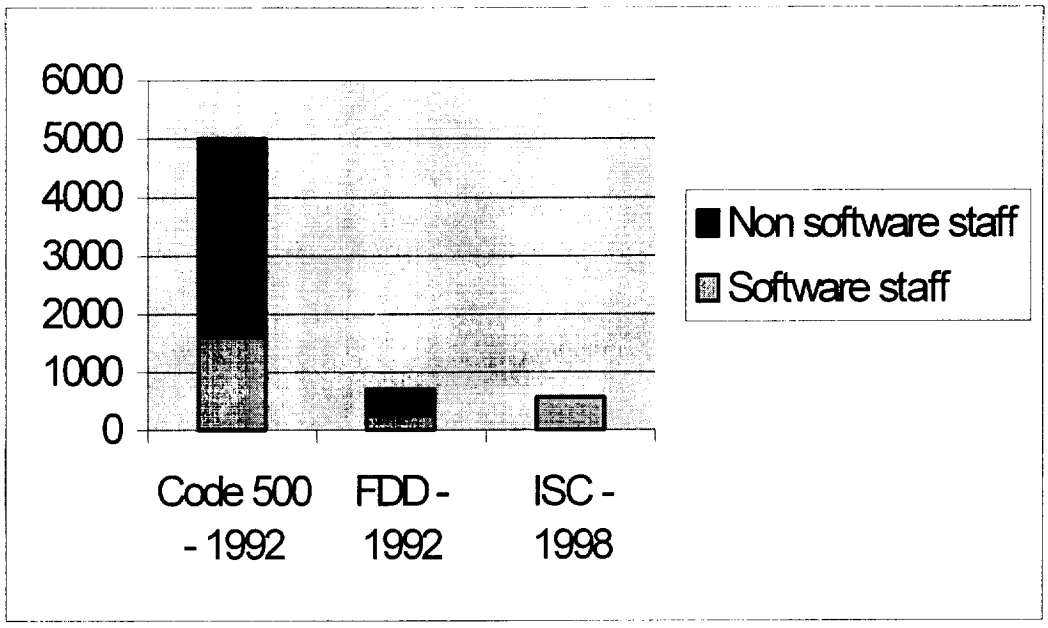


Figure 6 – Code 500, FDD and ISC staff

Branch management was also asked to estimate effort distribution within three categories: Development, Maintenance and Other. The results for this question are shown in Figure 7. The average is weighted for head-counts in the respective branches. Notable contrasts here are 581's large amount of "other" activity – as a systems integration management branch they do hardly any software development themselves. Also notable is 584 (Goddard real-time software)'s large maintenance effort relative to development effort, and 586 (science data systems)'s large development effort relative to maintenance.

In comparison with the code 500 baseline, maintenance effort in the code 500 was a lower proportion of total effort (24%) as opposed to ISC's 35% of effort devoted to maintenance. This is probably explained by the smaller amount of legacy code that the ISC is responsible for maintaining, in comparison to code 500.

Figure 8 turns our focus on software development effort alone, broken into the activities 'requirements analysis', 'design', 'coding', 'testing' and 'other'. It is apparent that at this macro process level, there is relatively little difference between ISC's average development effort distribution and that of the 1992 FDD. The ISC do a little more requirements, but that is the only major difference. Again, we should stress that these data are management estimates, not the actual recorded effort for each employee. In some cases, managers used heuristics such as counting the number of testers in the organization to come up with the proportion of testing being done. But did this then account for developers' unit testing? We do not know.

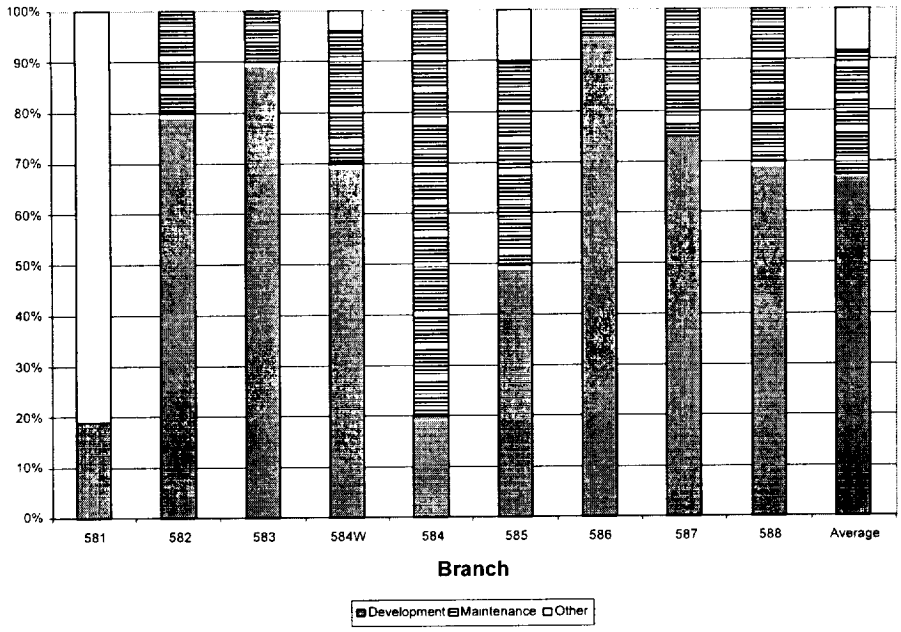


Figure 7 - Overall Effort by Branch

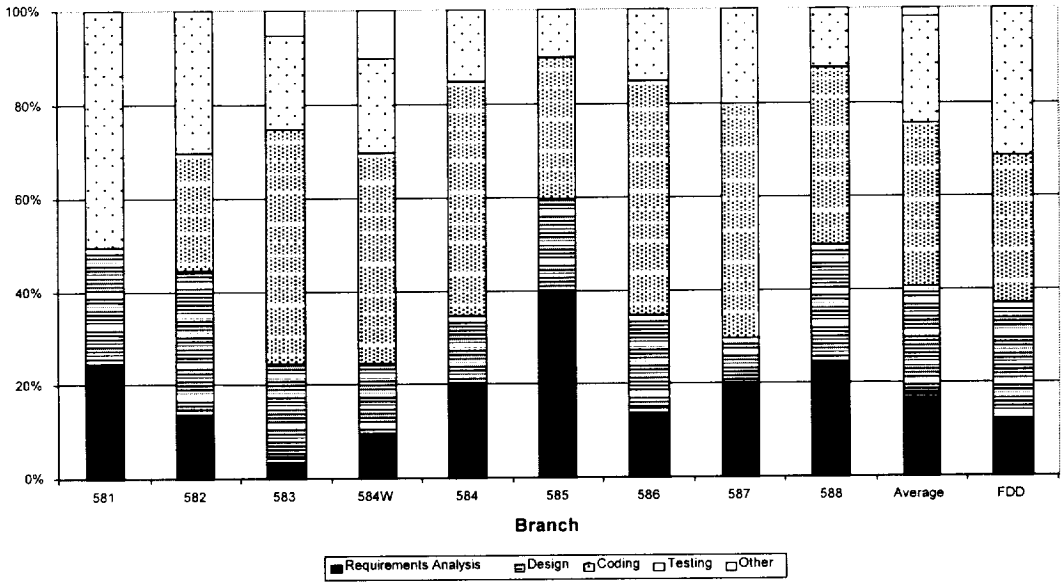


Figure 8 - Development Effort by Branch

One possible interpretation of this data is that organizations that are more outwardly focused, have had to put more effort into the requirements stage (and hence proportionally less in other areas). Code 585 (science data systems) is an example of this – much of their work is for the science community as a whole, a fairly diverse and remotely located user population. Code 583

(mission applications) has a much more defined user base and develops software such as off-line mission scheduling systems that can be precisely specified more easily up-front.

Some further observations about process, product and knowledge levels. Note that all branch averages are weighted by the number of staff in the branch.

- The percent of branches (including contractors) using "defined, written, advocated software processes" varied from 10-95%, with an average of 45%
- The percent of branches (including contractors) using "software standards" ranged from 0-95%, with an average of 57%
- The number of COTS products used varied from 2-10 with an average of 5.1. Note that these figures are probably deflated due to some branches listing "DBMSs", or "lots" in response to this question.
- Overall the use of C++, Java and Ada for new development is increasing, relative to Assembly, Fortran and C. 12 languages are used across ISC as a whole.
- The most significant causes of errors in operational software were (in the following order of importance): 'changing requirements', 'missing requirements', 'misinterpreted requirements', 'coding errors', 'interfaces', 'design errors' and 'environment problems'.
- Most branches consider themselves well-informed about 'prototyping', 'object-oriented technology', 'inspections/walkthroughs', and 'COTS Integration'
- Most branches consider themselves to have relatively little knowledge about 'formal methods' and 'defect causal analysis', except 586 science data systems
- Most branches consider themselves to have relatively little knowledge about 'information hiding' except 584W real-time systems (Wallops)
- All branches consider themselves to have relatively little knowledge about 'Cleanroom techniques'.
- Only three branches produce 'lessons learned' documents at the end of a project. Interestingly, one of these (584W) also produce a document called 'a day in the life' which serves to portray a typical day's activities for a developer. This is considered useful for training purposes.

In the process improvement area, several of the branches have ongoing activities:

- Code 581 is funding this ISC baselining study, and is also leading the ISO 9000 ISC certification. It is also pursuing an effort to define a core metrics set with the SEL and Code 300.
- Code 582 is encouraging reuse of both flight software and ground simulators, is looking into additional opportunities for automatic code generation, and is pursuing the use of COTS.
- Code 583 has implemented the CORE TEAM approach, which is a type of process improvement, and some parts of the branch are involved in some level of data collection.

- Codes 584 and 587 are currently defining their processes, as a prelude to improving them. Code 584 expressed a desire to define a multi-level process structure, to facilitate modularization of processes.
- Code 585, although it has not initiated a formal process improvement program, is using guidelines in certain areas. The Code 585 personnel prefer to use guidelines, rather than standards, because of the greater flexibility that guidelines provide.
- Code 586 is engaged in process management activities, including implementation of ISO 9001.
- Code 588, for the most part, has not initiated any process improvement activities; they are, however, currently working on a Technology Management Plan that is oriented toward ISO 9000. Code 588 is also trying to move the designation of their ultimate customer organization earlier in the process of making a system operational.

Analysis and further activities

The ISC is a new organization that supports many of the key projects at NASA Goddard. It is divided into management, technology and functional branches that represent a wide variety of technical and functional domains. Here we try to summarize the main results of the baselining effort and their implications for further SEL activities.

Diversity

The preliminary results of this baseline show that each branch is very different in terms of personnel, process and product characteristics. The variations in effort distribution, languages used, and products developed by the different branches provide surface indications of the diversity among the branches. The implications are that it will not be possible to apply the same models for cost and quality to each branch, as we could do to some extent within the more homogeneous FDD. To understand how cost and quality relate, we must study them in the context of each branch, team and/or project. Then, each model must be constructed and calibrated to the given context in question. The development of different models however is not the only challenge; these models must be capable of integration so that aggregated information can be meaningfully provided for the whole of ISC.

The NASA Core Software Metrics Initiative

The SEL and GSFC/NASA's Software Assurance Technology Center (SATC) [11] are currently pursuing an initiative to define and implement a core set of software metrics, common to the whole of NASA. For well over a year these two GSFC organizations have been working together to define a core set of metrics.

The baselining has confirmed that there is an essential need for core metrics within the ISC. Due to the diversity of the ISC, branches, teams and projects use different reporting units for metrics such as product size, effort and defects. The core metrics initiative defines a set of metrics capable of being used in different contexts, yet capable of providing a common abstraction level to allow aggregation at the ISC level. This is essential not only for monitoring purposes, but also for the model building needs mentioned above.

At this time, a draft version of the Core Metrics set, developed by the SEL and SATC, is currently under review by the NASA Software Working Group. At the time this paper is written the SATC and SEL web pages do not specifically call out the Core Metrics, in future that information should be assessable through SATC and SEL web pages [11,12]. An experiment within the ISC to validate these Core Metrics would serve both the NASA Core Metric Initiative and the ISC's proactive drive toward process and product awareness and improvement.

Matrixing

The ISC is organized in branches and teams, but branch and team staff work, at 63% on average, on projects outside the scope of ISC, managed and funded by NASA Codes other than 500. In particular, 95% of the staff belonging to Code 582 is matrixed outside ISC. This is not surprising, as the ISC is meant to offer IT services to all of GSFC and NASA. However, a number of issues are raised.

- System and software engineering. Many projects where matrixed staff works are system projects where software is only a part. The system issues (processes, technologies, interfaces) should be taken into account in software processes too.
- Ownership of processes and rights to modify. When projects are funded and ruled outside ISC, ISC may or may not be free to decide on processes, standards, and organizations to be used.
- Diffusion of information. Matrixed personnel could physically work outside ISC, with increased difficulties in communication and diffusion of information about the SEL and technology transfer or software process improvement projects.

The SEL could try to understand in more depth these issues with further studies. However, it seems that, for the purposes of assessment, characterization, and model building, the team and the projects are the more suitable units to be considered. This implies that, as projects and teams are volatile, with a life span of months, measures and models should be highly versatile and adaptive.

Also, the concept of Experience Factory, defined and used by the SEL in the past years, could need some adaptation. Several levels of experience, and several levels of learning loops, can be identified: at the individual, team, branch and ISC levels.

Finally, if projects and teams are volatile, and branches are virtual, individual persons are the most stable and valuable resources to base process and product improvement on. Approaches such as Watt Humphrey's Personal Software Process (PSP) could be used and adapted to the ISC context. Specifically, the PSP does not consider sharing experiences and improvements with peers, and should be extended in this direction to integrate concepts from the Experience Factory.

COTS

All branches report the use of COTS. The SEL should support teams and branches in COTS related activities: evaluation and selection, testing and certification, interaction with producer, documentation and diffusion of information. The SEL's experience in COTS processes will be of benefit to the ISC and the diversity of the ISC offers opportunities for case studies to further

validate the COTS process model [9]. This study concluded with recommendations for further work to build cost models, risk analysis, and process models. Since, COTS remains a buzzword with different meanings for different people. Another action for the SEL is the definition of a set of terms and classification tools for the different concepts and artifacts currently considered under the umbrella term COTS.

Finally, COTS should be considered in the broader context of reuse and related technological and organizational issues: domain analysis and engineering, product line engineering, reusable libraries, frameworks, design patterns, mechanisms and standards (Com, Corba, Active-X, Java RMI, Java beans, etc.).

Internal technology transfer

There would seem to be opportunities for greater synergies within ISC to do internal technology transfer so that the advanced technologies and research efforts of branches 585, 587 and 588 are successfully transitioned into practice in branches 582, 583, 584 and 586.

The past work of the SEL within Goddard has shown the need to understand, assess and package technology to insure its successful introduction. Possibly the SEL in code 581 can play a role in furthering a controlled and systematic transfer of this technology to the functional branches, as well as helping insure that the advanced technology branches work in relevant areas amenable to future technology transfer.

The SEL could assist by defining a methodology to evaluate if and how a technology successfully applied in one context (branch, team, project) can be transferred to another context.

Reuse and frameworks

Several products in ISC are developed and mantained for years and possibly customised in different versions. The overall cost of a product during the complete service cycle can be decreased by technologies such as architecture and framework-based reuse. For example Code 582 (flight software) is exploring this road by developing a new architectural design for on-board shuttle navigation control.

The SEL could offer support to organize, measure and document such efforts with two main goals. Promote the success of the reuse effort inside a branch. And acquire methodological experience to replicate the same effort in other branches (see also the Internal Technology Transfer subsection).

Requirements instability

Requirements, and specifically requirements instability, are a common source of problems for ISC teams. Several lines of intervention are available for the SEL:

- Experimentation with novel techniques for requirements capture and management.
- Adaptation of and experimentation with of techniques for early detection of defects in requirements, such as requirement reading techniques.
- Adaptation of and experimentation with new lifecycles for early verification of requirements, such as prototyping, iterative lifecycles, joint application development.

Acknowledgements

This work was funded by NASA grant NCC-5170, and the following NASA Contracts: CNMOS and CSOC.

References

[1] V. R. Basili, G. Caldiera, F. McGarry, R. Pajerski, G. Page, S. Waligora, The Software Engineering Laboratory - an Operational Software Experience Factory, *International Conference on Software Engineering*, May, 1992, pp. 370-381.

[2] R. Basili, H. D. Rombach, The TAME Project: Towards Improvement-Oriented Software Environments, *IEEE Transactions on Software Engineering*, vol. SE-14, no.6, June 1988.

[3] Kea H., Goddard's New Integrated Approach to Information Technology, 23rd Software Engineering Workshop, Nasa/GFSC, December 1998.

[4] NASA, Profile of Software Within Code 500 at Goddard Space Flight Center, Technical report R01-92, 1992.

[5] NASA, Profile of Software at the Goddard Space Flight Center, Technical report RPT-002-94, June 1994.

[6] NASA, Profile of Software at NASA, Technical report RPT-93, December 1993.

[7] NASA, An Overview of the Software Engineering Lab, Technical report SEL-94-005, December 1994.

[8] NASA/ISC, The ISC home page, http://isc.gsfc.nasa.gov/default.htm.

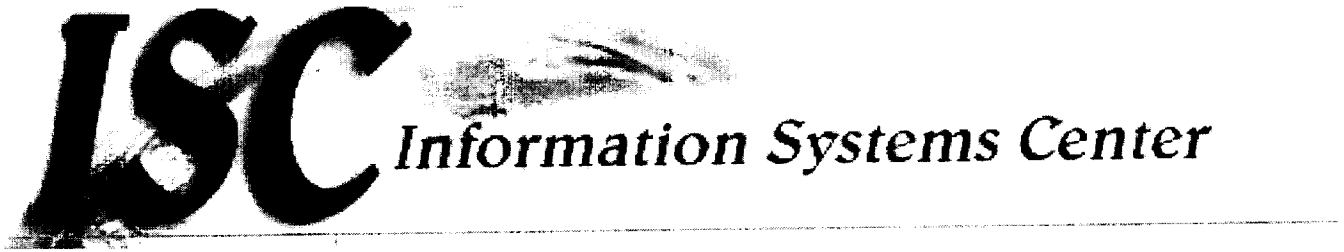
[9]NASA/SEL, SEL COTS Study, Phase 1, Initial Characterization Study report, SEL-98-001, August 1998.

[10] NASA/SEL, ISC Baselining documentation, http://sel.gsfc.nasa.gov/doc-st/tech-st/sew23/baselining.htm

[11] NASA/SATC, The SATC home page, http://satc.gsfc.nasa.gov/

[12] NASA/SEL, The SEL home page, http://sel.gsfc.nasa.gov/

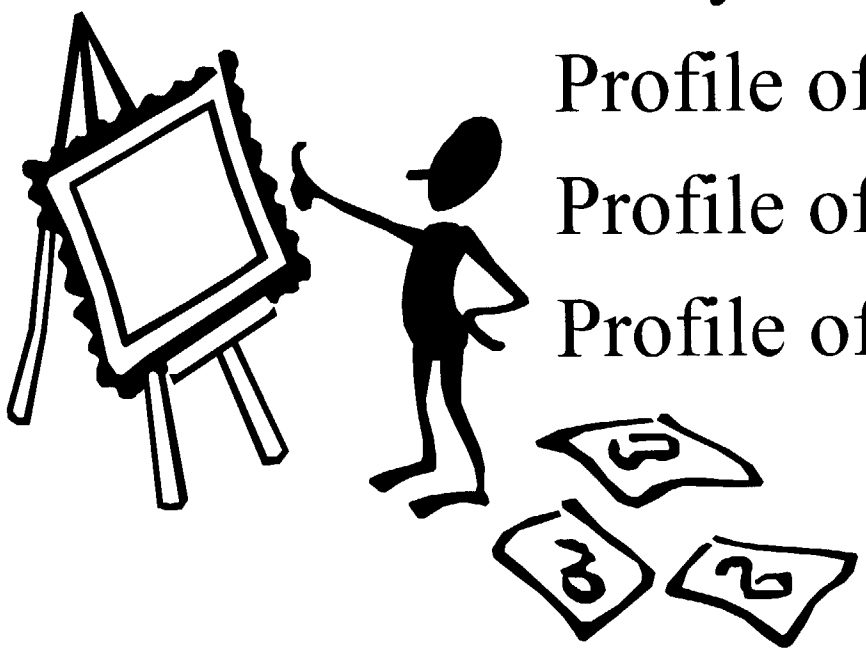
Baselining the New GSFC



The Foundation for Verifiable Software Process Improvement

A. Parra, D. Schultz, J. Boger, S. Condon, *CSC*
V. Basili, R. Webby, M. Morisio, D. Yakimovich, J. Carver, M. Stark, *U. of MD*
S. Kraft, *GSFC*

Prior Baseline Studies



Many Baselines of FDD

Profile of GSFC Code 500 (1993)

Profile of GSFC (1994)

Profile of NASA (1995)

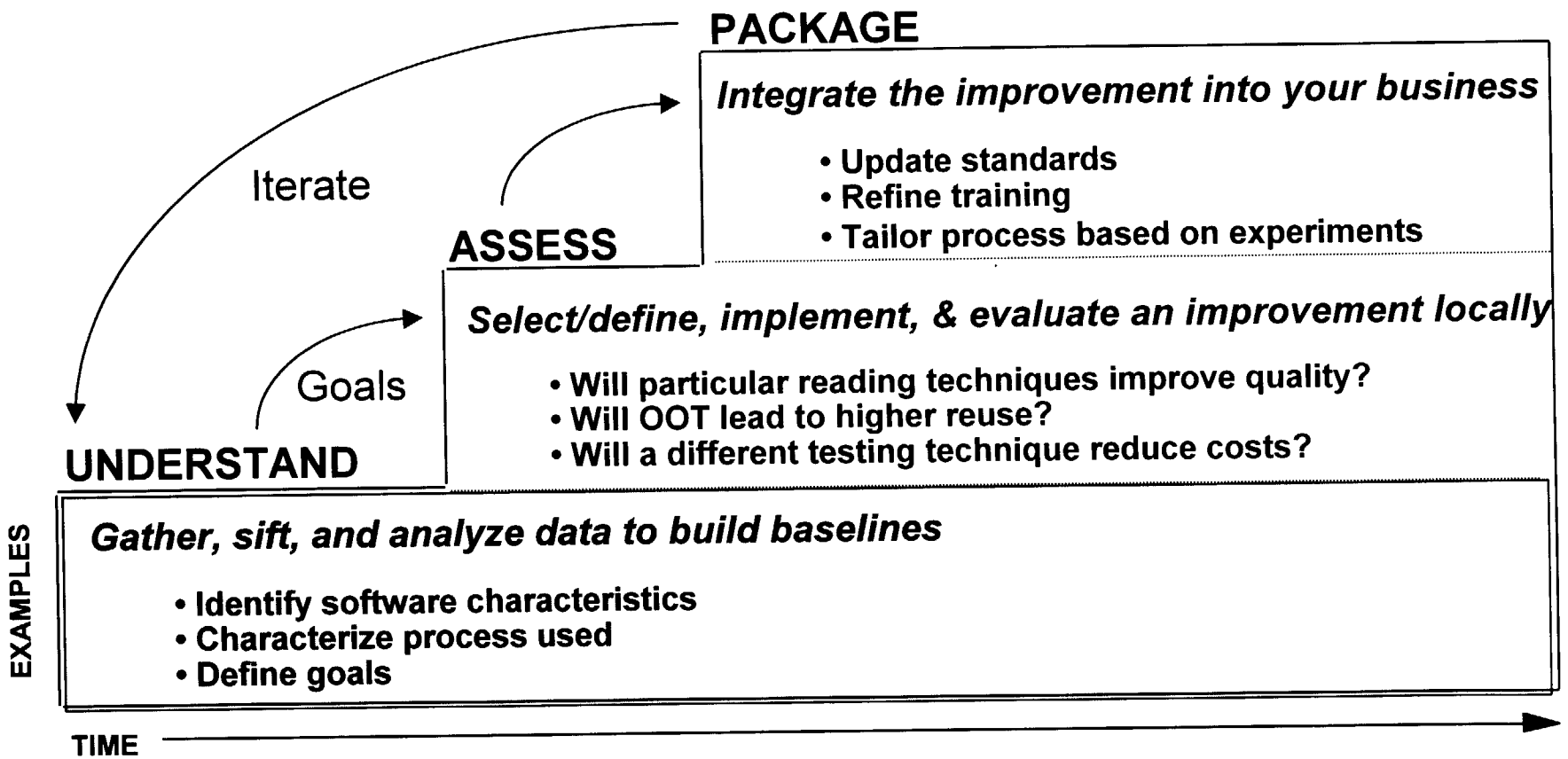
What's a Baseline?

organization
product
process

- Identify software characteristics
- Characterize process used
- Define goals
- Develop models to measure improvements

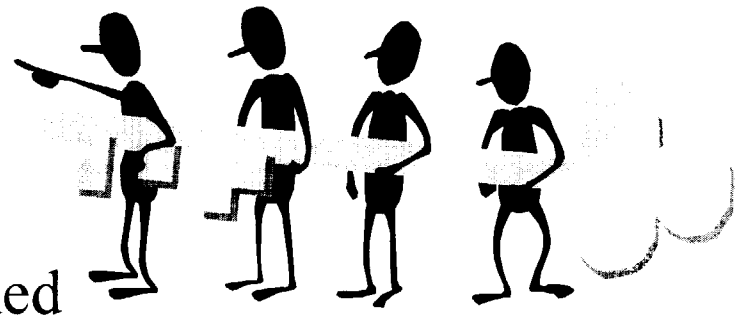
Gather, sift, & analyze data to build baselines

Role of Baselines in Process Improvement



Produce a baseline characterization - branches & teams

ISC Information Systems Center



Short term benefits -

share lessons learned

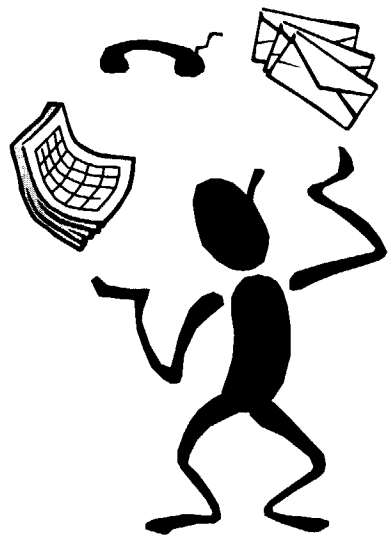
better understanding of the ISC

Long term benefits -

demonstrable product and process improvements

Methods of ISC Baseline Study

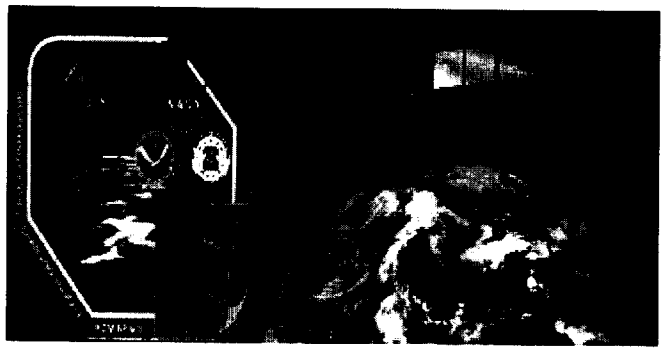
- Gather information at two levels
  - Branch
  - Team
- Gather information in two modes
  - Structured interviews
  - Questionnaires
- Analyze data
- Verify data & results with data provider



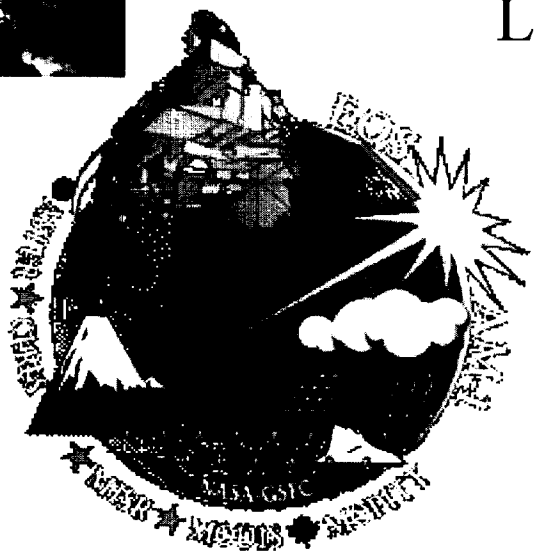
Next Steps for the Baselining

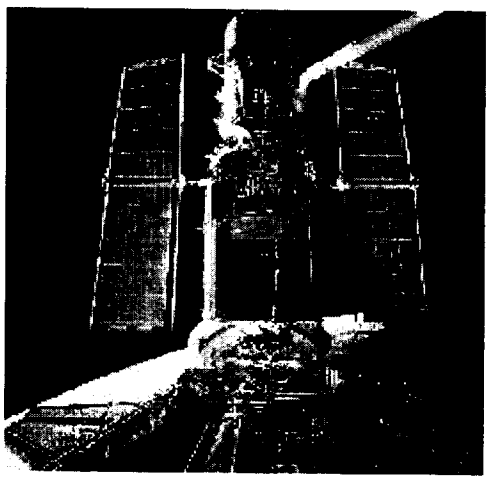
- Complete team level data extraction; analyze team level results
- Compare branch and team level data results
- Publish ISC baseline
- Study ISC environment
- Build models for ISC

The Diversity of ISC

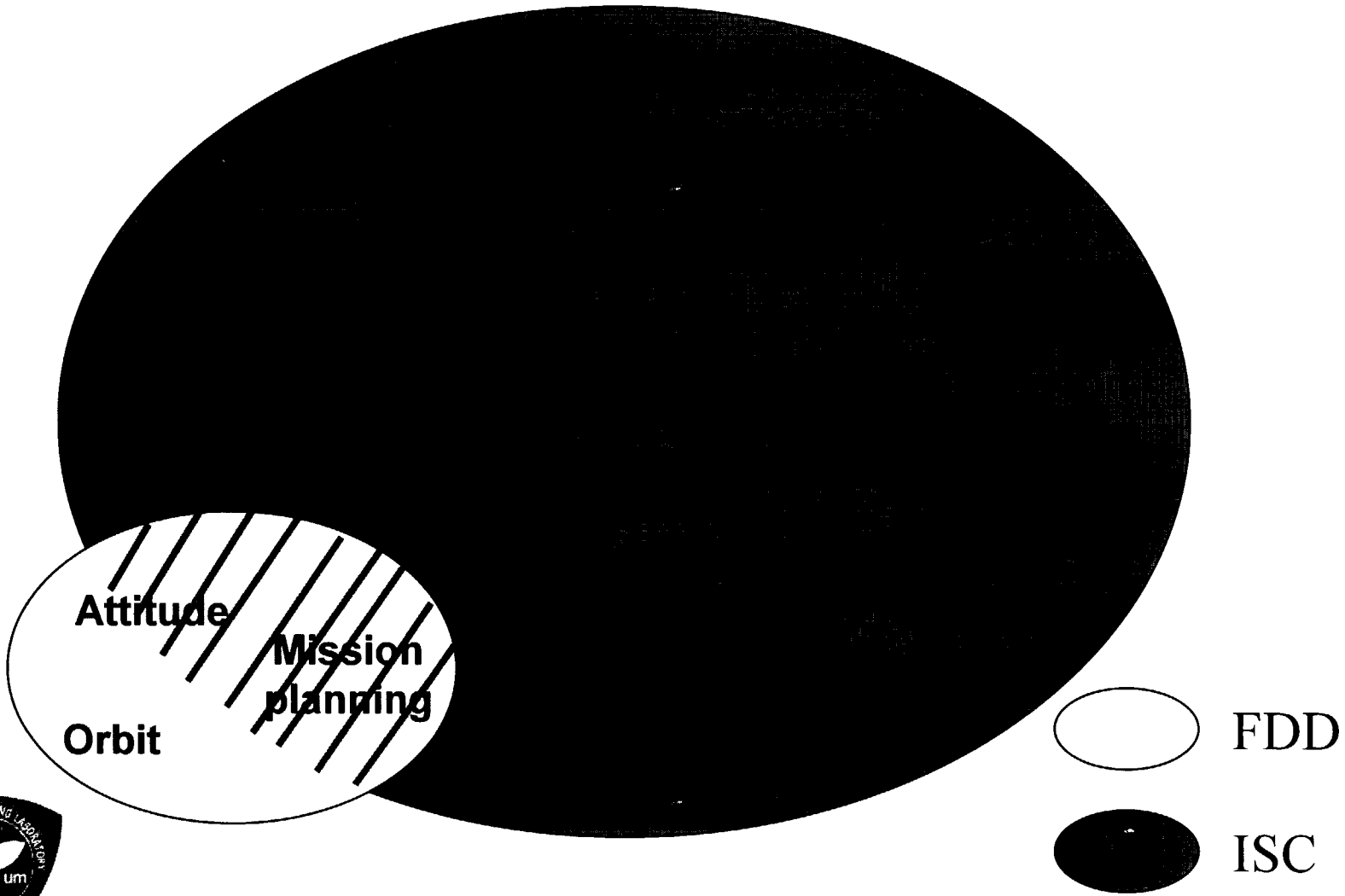


Application Domains
Matrixing
COTS Usage
Software Activities
Language

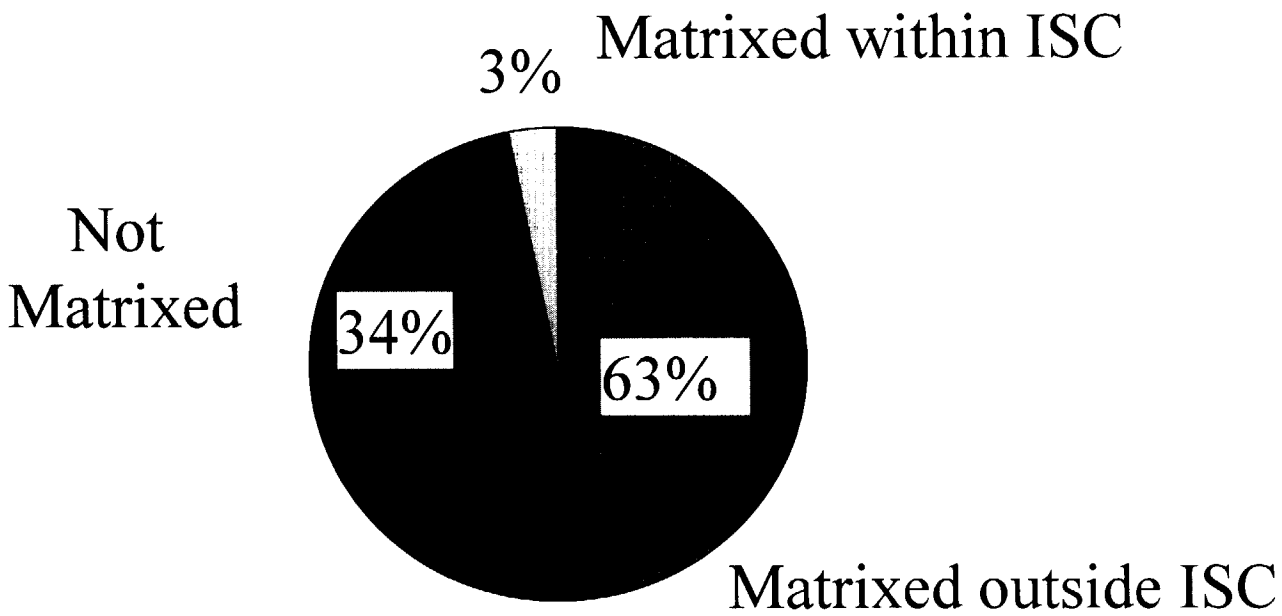


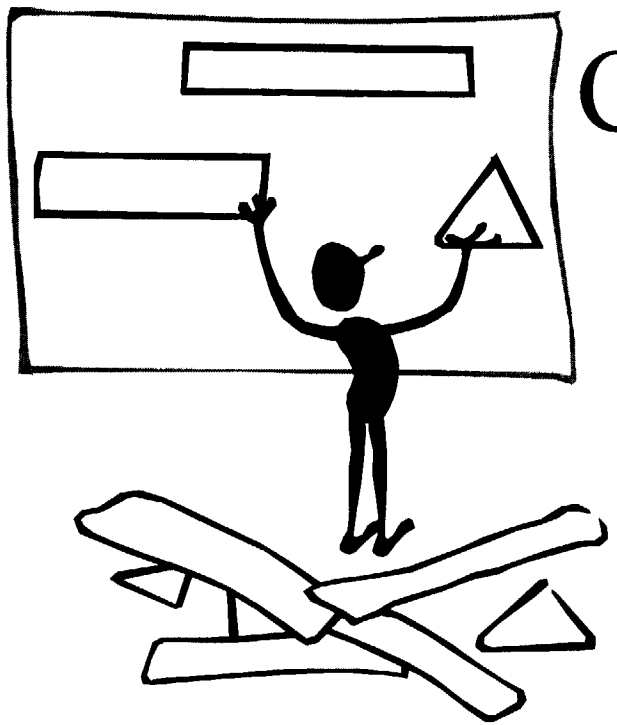


Application Domains



Matrixing

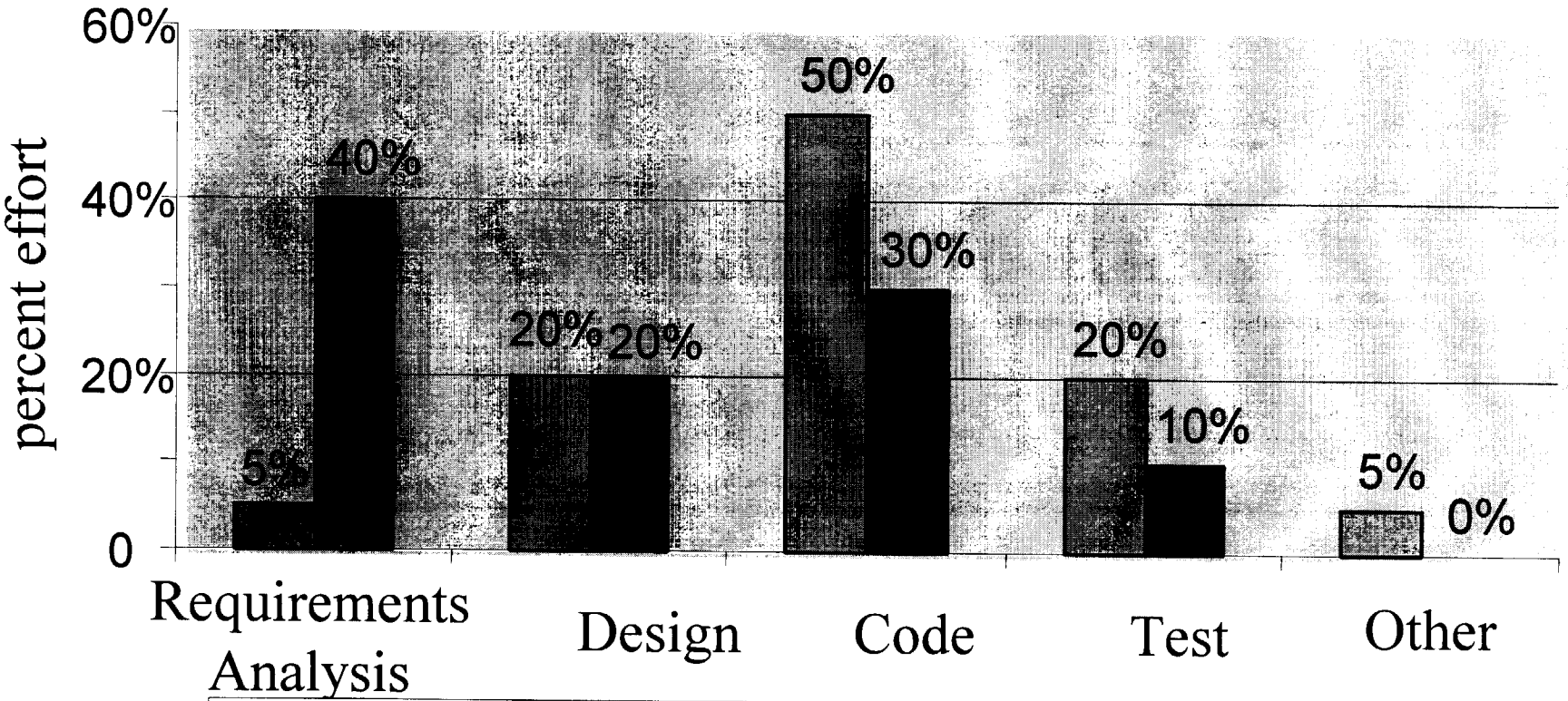

3%
Matrixed within ISC
Not Matrixed
34%
63%
Matrixed outside ISC



COTS Usage

- Heavy COTS usage across all branches
- Diverse COTS products support unique Application Domains

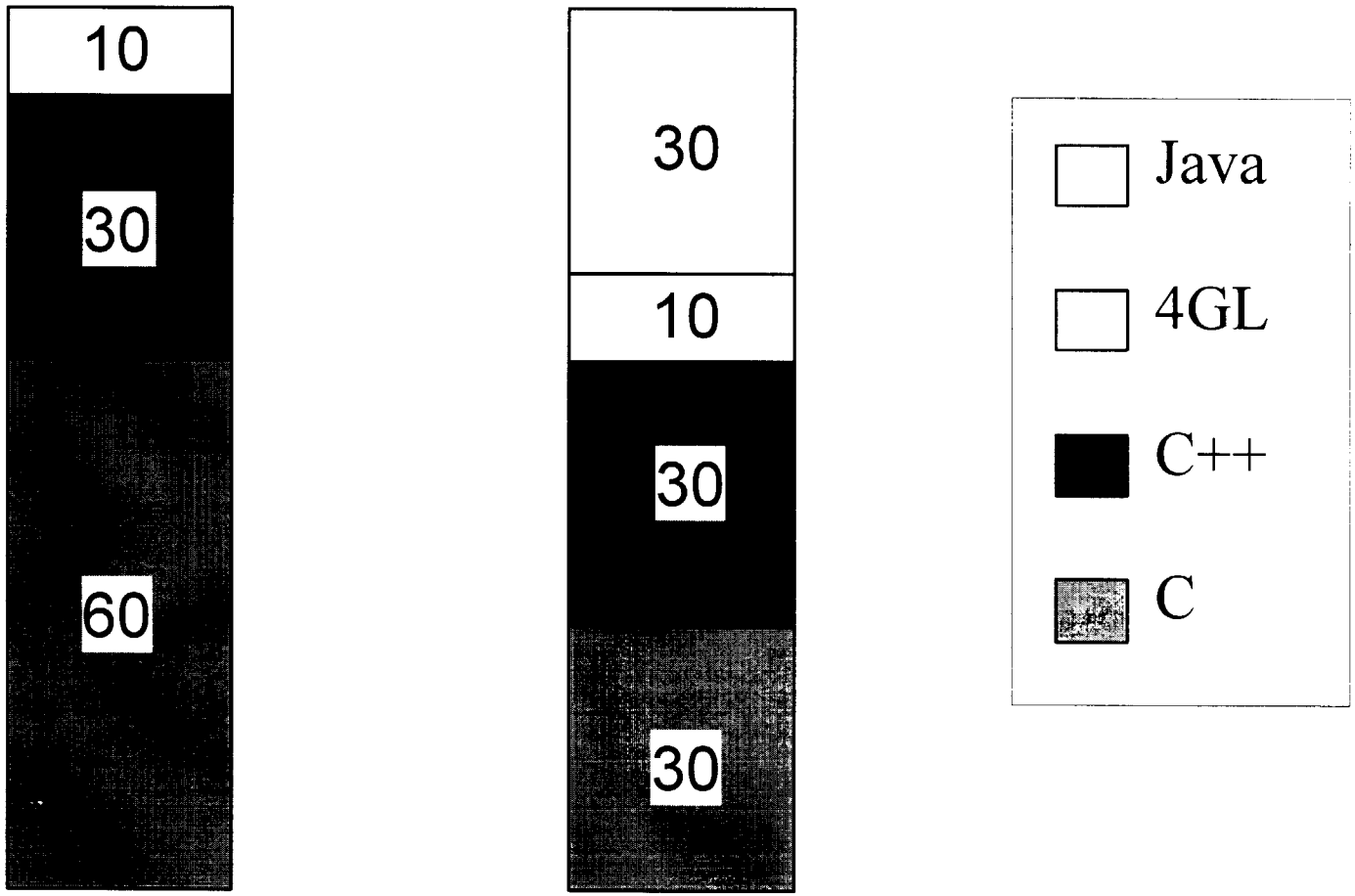
Software Activities



Branch A - 90% dev./ 10% maint.

Branch B - 50% dev./ 40% maint.

Language Mix



Existing Software

New Software

Initial Conclusions & Implications

- Diverse Domain - multiple models
- Matrixing - opportunity for Tech Transfer
- COTS - important issue, continue studies
- Software activities - multiple models
- Language Mix - evolution within a branch

Proposed Focus
based on Baselining Experience

- Cost estimation/defect models
- COTS Studies
- Process Improvement Program
  - To tailor & integrate in evolving organizations

S3-61

Using Experiments to Build a Body of Knowledge

Victor Basili
Fraunhofer Center Maryland
and Computer Science Dept.
University of Maryland
College Park, MD 20742, USA
basili@cs.umd.edu

Forrest Shull
Institute for Advanced Computer Studies
Computer Science Dept.
University of Maryland
College Park, MD 20742, USA
fshull@cs.umd.edu

Filippo Lanubile
Dipartimento di Informatica
Universita' di Bari
Via Orabona, 4
70126 Bari, Italia
lanubile@di.uniba.it

Abstract

Experimentation in software engineering is important but difficult. One reason it is so difficult is that there are a large number of context variables, and so creating a cohesive understanding of experimental results requires a mechanism for motivating studies and integrating results. This paper argues for the necessity of a framework for organizing sets of related studies. With such a framework, experiments can be viewed as part of common families of studies, rather than being isolated events. Common families of studies can contribute to important and relevant hypotheses that may not be suggested by individual experiments. A framework also facilitates building knowledge in an incremental manner through the replication of experiments within families of studies.

Building knowledge in this way requires a community of researchers that can replicate studies, vary context variables, and build abstract models that represent the common observations about the discipline. This paper also presents guidelines for lab packages, meant to encourage and support replications, that encapsulate materials, methods, and experiences concerning software engineering experiments.

1. Introduction

Experimentation in software engineering is necessary. Common wisdom, intuition, speculation and proofs of concepts are not reliable sources of credible knowledge. On the contrary, progress in any discipline involves building models that can be tested, through empirical study, to check whether the current understanding of the field is correct[1]. Progress comes when what is actually true can be separated from what is only believed to be true. To accomplish this, the scientific method supports the building of knowledge through an iterative process of model building, prediction, observation, and analysis. It requires that no confidence be placed in a theory that has not stood up to rigorous deductive testing [21]. That is, any scientific theory must be (1) falsifiable, (2) logically consistent, (3) at least as predictive as other competing theories, and (4) its predictions have been confirmed by observations during tests for falsification. According to Popper, a theory can only be shown to be false or not yet false; researchers only become confident in a theory when it has survived numerous attempts made at its falsification. This paradigm is a necessary step for ensuring that opinion or desire does not influence knowledge.

Experimentation in software engineering is difficult. Carrying out empirical work is complex and time consuming; this is especially true for software engineering. Unlike manufacturing, we do not build the same product, over and over, to meet a particular set of specifications. Software is developed and each

[1] For the purpose of this paper, we use the definitions of some key terms from [15] and [1]. An *empirical study*, in a broad sense, is an act or operation for the purpose of discovering something unknown or of testing a hypothesis, involving an investigator gathering data and performing analysis to determine what the data mean. This covers various forms of research strategies, including all forms of experiments, qualitative studies, surveys, and archival analyses. An *experiment* is a form of empirical study where the researcher has control over some of the conditions in which the study takes place and control over the independent variables being studied; an operation carried out under controlled conditions in order to test a hypothesis against observation. This term thus includes quasi-experiments and pre-experimental designs.
A *theory* is a possible explanation of some phenomenon. Any theory is made up of a set of hypotheses. A *hypothesis* is an educated guess that there exists (1) a (causal) relation among constructs of theoretical interest; (2) a relation between a construct and observable indicators (how the construct can be observed).
A *model* is a simplified representation of a system or phenomenon; it may or may not be mathematical or even formal; it can be a theory.

product is different from the last. So, software artifacts do not provide us with a large set of data points permitting sufficient statistical power for confirming or rejecting a hypothesis. Unlike physics, most of the technologies and theories in software engineering are human-based, and so variation in human ability tends to obscure experimental effects. Human factors tend to increase the costs of experimentation while making it more difficult to achieve statistical significance.

Abstracting conclusions from empirical studies in software engineering research is difficult. An important reason why experimentation in software engineering is so hard is that the results of almost any process depend to a large degree on a potentially large number of relevant context variables. Because of this, we cannot *a priori* assume that the results of any study apply outside the specific environment in which it was run. For isolated studies, even if they are themselves well-run, it is difficult to understand how widely applicable the results are, and thus to assess the true contribution to the field.

As an example, consider the following study:

- **Basili/Reiter**. This study was undertaken in 1976 in order to characterize and evaluate the development processes of development teams using a disciplined methodology. The effects of the team methodology were contrasted with control groups made up of development teams using an "ad hoc" development strategy, and with individual developers (also "ad hoc"). Hypotheses were proposed: that (BR1) a disciplined approach should reduce the average cost and complexity (faults and rework) of the process and (BR2) the disciplined team should behave more like an individual than a team in terms of the resulting product. The study addressed these hypotheses by evaluating particular methods (such as chief programmer teams, top down design, and reviews) as they were applied in a classroom setting. [7]

This study, like any other, required the experimenters to construct models of the processes studied, models of effectiveness, and models of the context in which the study was run. Replications that alter key attributes of these models are then necessary to build up knowledge about whether the results hold under other conditions. Unfortunately, in software engineering, too many studies tend to be isolated and are not replicated, either by the same researchers or by others. Basili/Reiter was a rigorous study, but unfortunately never led to a larger body of work on this subject. The specific experiment was not replicated, and the applicability of the hypotheses in other contexts was not studied. Thus it was never investigated whether the results hold, for example:

- for software developers at different levels of experience (the original experiment used university students);
- if development teams are composed differently (the original experiment used only 3-person teams);
- if another disciplined methodology had been used (i.e., were the benefits observed due to the particular methodology used in the experiment, or would they be observed for any disciplined methodology?).

2. A Motivating Example: Software Reading Techniques

Yet even when replications *are* run, it's hard to know how to abstract important knowledge without a framework for relating the studies. To illustrate, we present our work on reading techniques. Reading techniques are procedural techniques, each aimed at a specific development task, which software developers can follow in order to obtain the information they need to accomplish that task effectively [2, 3]. We were interested in studying reading techniques in order to determine if beneficial experience and work practices could be distilled into procedural form, and used effectively on real projects. We felt that reading techniques were of relevance and value to the software engineering community, since reading software documents (such as requirements, design, code, etc.) is a key technical activity. Developers are often called upon to read software documents in order to extract specific information for important software tasks, e.g. to read a requirements document in order to find defects during an inspection, or an Object-Oriented design in order to identify reusable components. However, while developers are usually taught how to *write* software documents, the skills required for effecting *reading* are rarely taught and must be built up through experience. In fact, we felt that research into reading could provide a model for how to effectively write documents as well: by understanding how readers perform more effectively it may be possible to write documents in a way that facilitates the task.

However, the concept of reading techniques cannot be studied in isolation. Like any other software process, reading techniques must be tailored to the environment in which they are run. Our aim in this research was to generate sets of reading techniques that were procedurally defined, tailorable to the environment, aimed at accomplishing a particular task, and specific to the particular document and notation on which they would be applied. This has led a series of studies in which we evaluated the following types of reading techniques:

- Defect-Based Reading (**DBR**) focused on defect detection in requirements, where the requirements were expressed using a state machine notation called SCR [13, 22].
- Perspective-Based Reading (**PBR**) also focused on defect detection in requirements, but for requirements expressed in natural language [4, 16].
- Use-Based Reading (**UBR**) focused on anomaly detection in user interfaces [27].
- Second Version of PBR (**PBR2**) consisted of new techniques that were more procedurally-oriented versions of the earlier set of PBR techniques. In particular, we made the techniques more specific in all of their steps [24].
- Scope-Based Reading (**SBR**) consisted of two reading techniques that were developed for learning about an Object-Oriented framework in order to reuse it [10, 23].

A framework that makes explicit the different models used in these experiments would have many benefits. Such a framework would document the key choices made during experimental design, along with their rationales. The framework could be used to choose a focus for future studies: i.e., help determine the important attributes of the models used in an experiment, and which should be held constant and which varied in future studies. The ultimate objective is to build up a unifying theory by creating a list of the specific hypotheses investigated in an area, and how similar or different they all are.

Using an organizational framework also allows other experimenters to understand where different choices could have been made in defining models and hypotheses, and raises questions as to their likely outcome. Because these frameworks provide a mechanism by which different studies can be compared, they help to organize related studies and to tease out the true effects of both the process being studied and the environmental variables.

3. The GQM Goal Template as a Tool for Experimentation

Examples of such organizational frameworks do exist in the literature, e.g. [9, 17, 20]. For the purpose of this paper we find the Goal/Question/Metric (GQM) Goal Template [8] useful. The GQM method was defined as a mechanism for defining and interpreting a set of operational goals using measurement. It represents a top-down systematic approach for tailoring and integrating goals with models of software processes, products, and quality perspectives, based upon the specific needs of a project and organization.

The GQM goal template is a tool that can be used to articulate the purpose of any study. It ties together the important models, and provides a basis against which the appropriateness of a study's specific hypotheses, and dependent and independent variables, may be evaluated. There are five parameters in a GQM goal template:

- *object of study*: a process, product or any other experience model
- *purpose*: to characterize (what is it?), evaluate (is it good?), predict (can I estimate something in the future?), control (can I manipulate events?), improve (can I improve events?)
- *focus*: model aimed at viewing the aspect of the object of study that is of interest, e.g., reliability of the product, defect detection/prevention capability of the process, accuracy of the cost model
- *point of view*: e.g., the perspective of the person needing the information, e.g., in theory testing the point of view is usually the researcher trying to gain some knowledge
- *context*: models aimed at describing environment in which the measurement is taken

For example, the goal of the Basili/Reiter study, previously described, might be instantiated as:

To analyze the *development processes of a 1) disciplined-methodology team approach, 2) ad hoc team approach, and 3) ad hoc individual approach*
for the purpose of *characterization and evaluation*

with respect to *cost and complexity (faults and rework) of the process*
from the point of view of the *developer and project manager*
in the context of *an advanced university classroom*

Due to the nature of software engineering research, instantiated goals tend to show certain similarities. The *purpose* of studies is often evaluation; that is, researchers tend to study software technologies in order to assess their effect on development. For our purposes, the *point of view* can be considered to be that of the researcher or knowledge-builder. While studies can be run from the point of view of the project manager, i.e. requiring some immediate feedback as to effects on effort and schedule, published studies have usually undergone additional, post-hoc analysis.

The remaining fields in the template require the construction of more complicated models, but still show some similarities. The *object of study* is often (but not always) a process; researchers are often concerned with evaluating whether or not a particular development process represents an improvement to the way software is built. (E.g.: Does Object-Oriented Analysis lead to an improved implementation? Does an investment in reviews lead to less buggy, more reliable systems? Does reuse allow quality systems to be built more cheaply?) When the object of study is a process, the *focus* of the evaluation is the process' effect. The experimenter may measure its effect on a product, that is, whether the process leads to some desired attribute in a software work product. Or, the experimenter may attempt to capture its effect on people, e.g. whether practitioners were comfortable executing the process or found it tedious and infeasible. Finally, the *context* field should include a large number of environmental variables and therefore tends to exhibit the most variability. Studies may be run on students or experts; under time constraints, or not; in well-understood application domains, or in cutting-edge areas. There are numerous such variables that may influence the results of applying a technique.

For the remainder of this paper, we will illustrate our conclusions by concentrating on studies that investigate process characteristics with respect to their effects on products. A GQM template for this class of studies is:

> Analyze processes to evaluate their effectiveness on a product from the point of view of the knowledge builder in the context of (a particular variable set).

For particular studies in this class, constructing a complete GQM template requires making explicit the process (object of study), the effect on the product (focus), and context models in the experiment. Making these models explicit is necessary in order to understand the conditions under which the experimental results hold.

For example, consider the GQM templates for the list of reading technique experiments described in the previous section. There are many ways of classifying processes, but we might first classify processes by scope as:

- Techniques (processes that can be followed to accomplish some specific task),
- Methods[2] (processes augmented with information concerning when and how the process should be applied),
- Life Cycle Models (processes which describe the entire software development process).

Each of these categories could be subdivided in turn. The set of techniques, for example, could be classified based on the specific task as: Reading, Testing, Designing, and so on. We have found it helpful to think of the range of values as organized in a hierarchical fashion, in which more general values are found at the top of the tree, and each level of the tree represents a new level of detail. (Figure 1)

Selecting a particular type of process for study, our GQM template then becomes:

> Analyze reading techniques to evaluate their effectiveness on a product from the point of view of the knowledge builder in the context of a particular variable set

[2] The definitions of "technique" and "method" are adapted from [5].

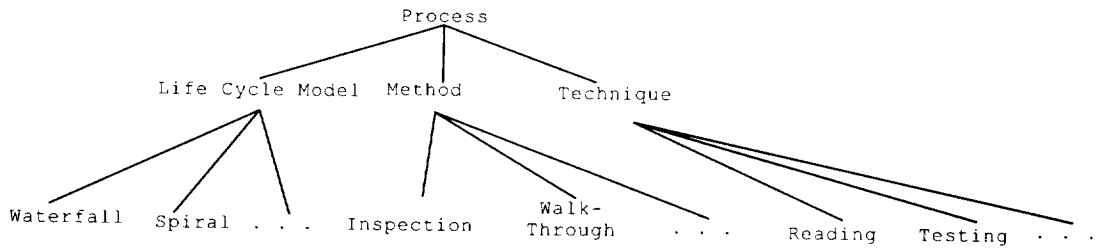


Figure 1: A portion of the hierarchy of possible values for describing software processes.

The reading technique experiments were concerned with studying the effect of the reading technique on a product. So, the model of focus needs to specify both how effectiveness is to be measured and the product on which the evaluation is performed.We find it useful to divide the set of effectiveness measures into analysis and construction measures, based on whether the goal of the process is to analyze intrinsic properties of a document or to use it in building a new system. Each of these categories can be further broken down into more specific types of process goals, for which different effectiveness measures may apply (Fig. 2). For example, the effectiveness of a process for performing maintenance can be evaluated by how that process effects the cost of making a change to the system. The effectiveness of a process for detecting defects in a document can be measured by the number of faults it helps find. Of course, many more measures exist than will fit into Figure 2. For instance, rather than measure the number of faults a defect detection process yields, it might be more appropriate to measure the number of errors[3], or the amount of effort required, among other things.

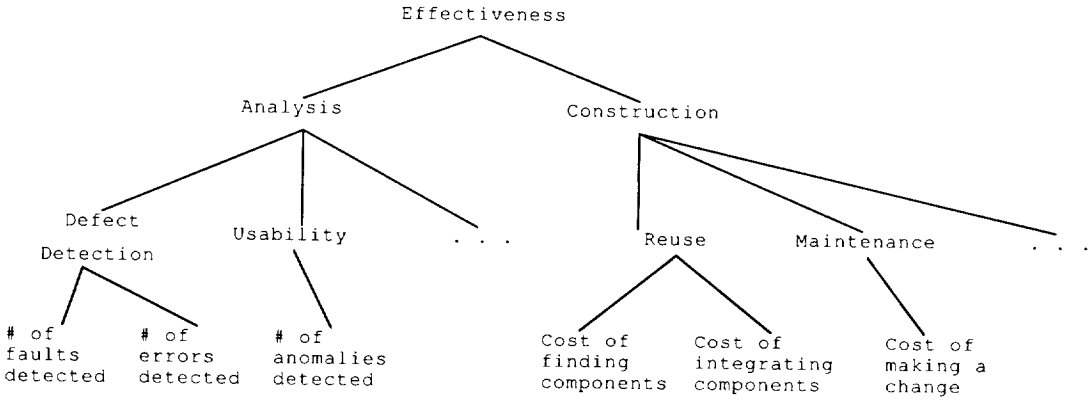


Figure 2: A portion of the hierarchy of possible values for describing the effectiveness of software processes

Similarly, a software document can be classified according to the model of a software system it contains (a relatively well-defined set) and further subdivided into the specific notations that may be used (Fig.3). The main purpose of organizing the possible values hierarchically is to organize a conception of the problem space that can be used by others for classifying their own experiments. The actual criteria used are somewhat subjective; naturally there are multiple criteria for classifying processes, effectiveness measures, and software documents, but we have selected just those that have contributed to our conception of reading techniques.

[3] Here we are using the terms "faults" and "errors" according to the IEEE standard definitions [14], in which "fault" refers to defects appearing in some artifact while "error" refers to an underlying human misconception that may be translated into faults.

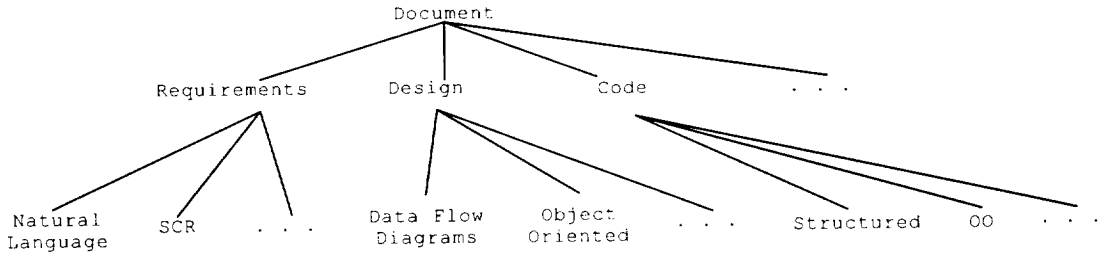


Figure 3: A portion of the hierarchy of possible values for describing software documents.

Thus a GQM template for the PBR experiment could be:

> Analyze reading techniques to evaluate their ability to detect defects in a Requirements Document written in English from the point of view of the knowledge builder in the context of a particular variable set .

A GQM goal is not meant to be a definitive description, but reflects the interests and priorities of the experimenter. If we were to study the process model for the reading techniques in each experiment in more detail, we would see that each technique is tailored to a specific task (e.g., analysis or construction, etc.) and to a specific document. This is what characterizes the reading techniques and distinguishes them from one another. Thus the process goals used to classify measures of effectiveness in Figure 2 can be easily adapted to describe the processes themselves (Figure 4). The distinction between analysis and construction process goals can apply directly to processes. That is, we hypothesize that analysis tasks differ sufficiently from construction tasks that, along with differences in the way they may be evaluated for effectiveness, there may also be different guidelines used in their construction. Thus figures 2 and 3 can also be mechanisms for identifying process model attributes. They should be accounted for in the process model as well as the effect on process.

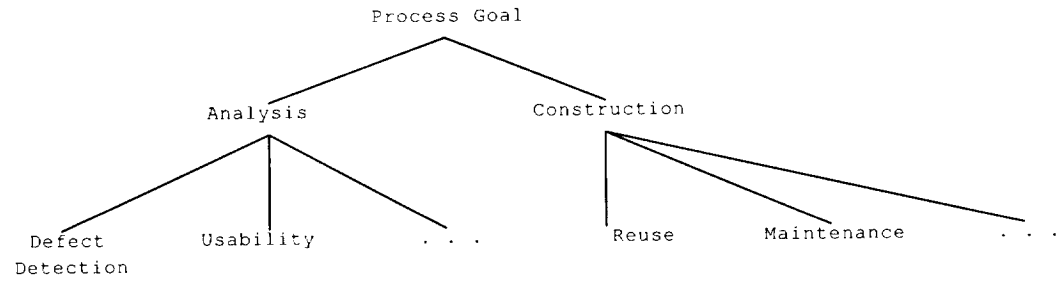


Figure 4: A portion of the hierarchy of possible values for describing the goal of a software engineering process.

Thus we can say that we are:

> analyzing a reading technique *for the purpose of* evaluating its ability to detect defects in a natural language requirements document

or we can say that we are:

> analyzing a reading technique *tailored to* defect detection in natural language requirements for the purpose of evaluation.

It depends on whether we are emphasizing the definition of the process or of its effectiveness.

In linking goal templates to hypotheses, we can think of the process model (object of study) as the independent variable, the effect on product (focus) as the dependent variable, and the context variables as the variables that exist in the environment of the experiment. The differences or similarities between experimental hypotheses can then be described in terms of these hierarchies of possible values. For example, consider the studies of DBR and PBR. In both cases, the process model was focused on the same

task (defect detection); although the notation differed, both were also focused on the same document (requirements). If all other attributes for process, product, and context models were held constant, we could begin to think of hypotheses at a higher level of abstraction. That is, instead of the hypothesis:

> **Subjects using a reading technique tailored to defect detection in natural language requirements are more effective than subjects using ad hoc techniques for this task**

The following hypothesis might be more useful:

> **Subjects using reading techniques tailored to defect detection in requirements are more effective than subjects using ad hoc techniques for this task.**

The difference between these hypotheses is that the focus of the study is described at a higher level of abstraction for the second hypothesis (requirements) than for the first (natural language requirements).

This difference in abstraction makes the second hypothesis more difficult to test. In fact, probably no single study could ever give us overwhelming evidence as to its validity, or lack thereof. Testing the second hypothesis would require some idea of what types of requirements notation are of interest to practitioners. Building up a convincing body of evidence requires the combined analysis of multiple studies of specific reading techniques for defect detection in requirements. But the effort required to formulate the hypothesis and begin building a body of evidence helps advance the field of software engineering. At best, the evidence can lead to the growth of a body of knowledge, containing basic and important theories underlying some aspect of the field. At worst, the effort spent in specifying the models forces us to think more deeply about the relevant ways of characterizing software engineering models that we, as researchers, are implicitly constructing anyway.

The above discussion should not be taken to imply that the attributes identified in Figures 1 through 4 are the only ones that are important, or for which hierarchies of possible values exist. To choose another example, in specifying the model of the context it is almost always important to characterize the experience of the subjects of the experiment. The most appropriate way of characterizing experience depends on many things; two possibilities are proposed in Figure 5.

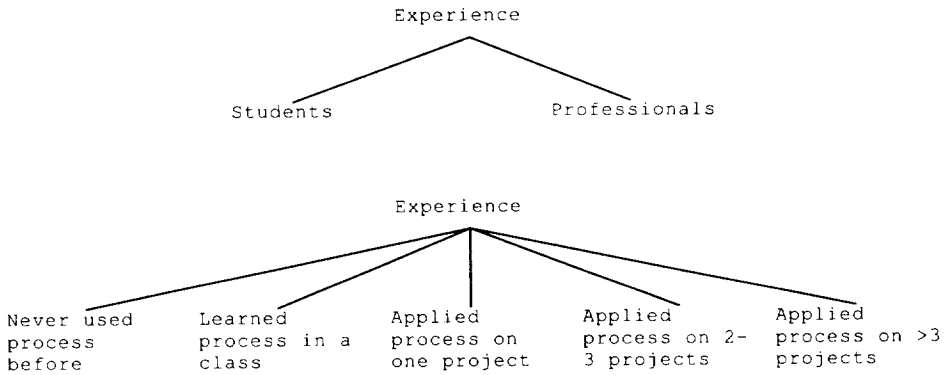


Figure 5: Two possible value hierarchies for measuring subject experience.

The trees shown in Figure 5 present two different ways of characterizing experience. The first is a simpler way of characterizing the attribute that distinguishes only between subjects who are still learning software engineering principles versus those who have applied them on real projects. The second hierarchy attempts to place finer distinctions on the amount of experience a subject has applying a particular process. Each may be appropriate to different circumstances.[FS1]

4. Replicating Experiments

In preceding sections of this paper, we have tried to raise several reasons why families of replicated experiments are necessary for building up bodies of knowledge about hypotheses. Another reason for running replications is that they can increase the amount of confidence in results by addressing certain threats to validity: Internal validity defines the degree of confidence in a cause-effect relationship between factors of interest and the observed results, while external validity defines the extent to which the

conclusions from the experimental context can be generalized to the context specified in the research hypothesis [11]. In this section, we discuss replications in more detail and look at the practical considerations that result.

Our primary strategy for supporting replications in practice has been the creation of lab packages, which collect information on an experiment such as the experimental design, the artifacts and processes used in the experiment, the methods used during the experimental analysis, and the motivation behind the key design decisions. Our hope has been that the existence of such packages would simplify the process of replicating an experiment and hence encourage more replications in the discipline. Several replications have been carried out in this manner and have contributed to a growing body of knowledge on reading techniques.

4.1. Types of Replications

Since we consider that replications may be undertaken for various reasons, we have found it useful to enumerate the various reasons, each of which has its own requirements for the lab package. In our view the types of replications that need to be supported can be grouped into 3 major categories:

1. **Replications that do not vary any research hypothesis.** Replications of this type vary none of the dependent or independent variables of the original experiment.
   1.1. **Strict replications** (i.e. replications that duplicate as accurately as possible the original experiment). These replications are necessary to increase confidence in the validity of the experiment. They demonstrate that the results from the original experiment are repeatable, and have been reported accurately by the original experimenters.

   1.2. **Replications that vary the manner in which the experiment is run.** These studies seek to increase our confidence in experimental results by addressing the same problem as previous experiments, but altering the details of the experiment so that certain internal threats to validity are addressed. For example, a replication may vary the order of activities to avoid the possibility that results depend not on the process used, but on the order in which activities in the experiment are completed.

   The attempt to compensate for threats to internal validity may also lead to other types of changes. For example, a process may be modified so that the researchers can assess the amount of process conformance of subjects. Although the aim of the change may have been to address internal validity, the new process should be evaluated in order to understand whether unanticipated effects on process effectiveness have resulted. Thus such a replication would fall into the second major category, discussed below.

2. **Replications that vary the research hypotheses.** Replications of this type vary attributes of the process, product, and context models but remain at the same level of specificity as the original experiment.
   2.1. **Replications that vary variables intrinsic to the object of study (i.e. independent variables).** These replications investigate what aspects of the process are important by systematically varying intrinsic properties of the process and examining the results. This type of experiment requires the process to be supplied in sufficient detail that changes can be made. This implies that the original experimenters must provide the rationales for the design decisions made as well as the finished product. For example, researchers may question whether the specificity at which the process is described affects the results of applying the process. In this sense, the study of PBR2 may be seen as a replication of the study of PBR, in which the level of specificity of the process was varied but all other attributes of the process model remained the same.

   2.2. **Replications that vary variables intrinsic to the focus of the evaluation (i.e. dependent variables).** Replications of this type may vary the ways in which effectiveness is measured, in order to understand for what dimensions of a task a process results in the most gain. For example, a replication might choose another effectiveness measure from those listed in Figure 2, investigating whether a defect detection process is more beneficial for finding errors than faults.

Other aspects of the focus model might be varied instead, e.g. a process might be evaluated on a document of the same type but different notation to see if it is equally effective (see Figure 3).

2.3. **Replications that vary context variables in the environment in which the solution is evaluated.** These studies can identify potentially important environmental factors that affect the results of the process under investigation and thus help understand its external validity. For example, replications may be run using the same process and product models as the original experiment but on professionals instead of students (see Figure 5) to see if the same results are obtained.

3. **Replications that extend the theory.** These replications help determine the limits to the effectiveness of a process, by making large changes to the process, product, and/or context models to see if basic principles still hold. We discussed replications in the previous category as replacing the value of some variable (e.g. document on which the process was applied, Figure 3) with another, equally specific value (e.g. SCR requirements instead of English-language requirements). Replications in this category, however, can be thought of as replacing an attribute of a process, product, or context model with a value at a higher level of abstraction (i.e. from a higher level in the hierarchy). Again using Figure 3, researchers may choose to study whether a type of process is applicable to requirements documents in general, rather than limiting their scope to a specific kind. The type of hypotheses associated with such replications was discussed in section 3.

4.2 Implications for Lab Package Design

In software engineering research, there has been a movement toward the reuse of physical artifacts and concrete processes between experiments. This is indeed a useful beginning. The cost of an experiment is greatly increased if the preparation of multiple artifacts is necessary. Creating artifacts which are representative of those used in real development projects is difficult and time consuming. Reusing artifacts can thus reduce the time and cost needed for experimentation. A more significant benefit is that reuse allows the opportunity to build up knowledge about the actual use of particular, non-trivial artifacts in practice. Thus replications (and experimentation in general) could be facilitated if there were repositories of reusable artifacts of different types (e.g. requirements) which have a history of reuse and which, therefore, are well understood. (A model for such repositories could be the repository of system architectures [12], where the relevant attributes of each design in the repository are known and described.)

A first step towards this goal is the construction of web-based laboratory packages. At the most basic level, these packages allow an independent experimenter to download experimental materials, either for reuse or for better understanding. In this way, these packages support strict replications (as defined in section 4.1), which require that the processes and artifacts used in the original experiment be made available to independent researchers.

However, web-based lab packages should be designed to support more sophisticated types of replications as well. For example, packages should assist other experimenters in understanding and addressing the threats to validity in order to support replications that vary some aspects of the experimental setup. Due to the constraints imposed by the setting in which software engineering research is conducted, it is almost never possible to rule out every single threat to validity. Choosing the "least bad" set of threats given the goal of the experiment is necessary. Lab packages need to acknowledge this fact and make the analysis of the constraints and the threats to validity explicit, so that other studies may use different experimental designs (that may have other threats to validity of their own) to rule out these threats.

Replications that seek to vary the detailed hypotheses have additional requirements if the lab package is to support them as well. For example, in order for other experimenters to effectively vary attributes of the object of study, the original process must be explained in sufficient detail that other researchers can draw their own conclusions about key variables. Since it is unreasonable to expect the original experimenters to determine all of the key variables *a priori*, lab packages must provide rationales for key experimental context decisions so that other experimentalists can determine feasible points of variation of interest to themselves. Similarly, lab packages must specify context variables in sufficient detail that feasible changes

to the environment can be identified and hypotheses made about their effects on the results.

Finally, in order to build up a body of knowledge about software engineering theories, researchers should know which experiments have been run that offer related results. Therefore, lab packages for related experiments should be linked, in order to collect different experiments that address different areas of the problem space, and contribute evidence relevant to basic theories. The web is an ideal medium for such packages since links can be added dynamically, pointing to new, related lab packages as they become available. Thus it is to be hoped that lab packages are "living documents" that are changed and updated to reflect our current understanding of the experiments they describe.

Lab packages have been our preferred method for facilitating the abstraction of results and experiences from series of well-designed studies. Interested readers are referred to existing examples of lab packages: [25, 26]. By collecting detailed information and results on specific experiments, they summarize our knowledge about specific processes. They record the design and analysis methods used and may suggest new ones. Additionally, by linking related studies they can help experimenters understand what factors do or do not impact effectiveness.

4.3. The Experimental Community

A group of researchers, from both industry and academia, has been organized since 1993 for the purpose of facilitating the replication of experiments. The group is called ISERN, the International Software Engineering Research Network, and includes members in North America, Europe, Asia, and Australia. ISERN members publish common technical reports, exchange visitors, and organize annual meetings to share experiences on software engineering experimentation[4]. They have begun replicating experiments to better understanding the success factors of inspection and reading.

The *Empirical Software Engineering* journal has also helped build an experimental community by providing a forum for publishing descriptions of empirical studies and their replications. An especially noteworthy aspect of the journal is that it is open to publishing replicated studies that, while rigorously planned and analyzed, yield unexpected results that did not confirm the original study. Although it has traditionally been difficult to publish such "unsuccessful" studies in the software engineering literature, this knowledge must be made available if the community is to build a complete and unbiased body of knowledge concerning software technologies.

5. Conclusions

The above discussion leads us to propose that the following criteria are necessary before we can begin to build up comprehensive bodies of knowledge in areas of software engineering:

1. Hypotheses that are of interest to the software engineering community and are written in a context that allow for a well defined experiment;
2. Context variables, suggested by the hypotheses, that can be changed to allow for variation of the experimental design (to make up for validity threats) and the context of experimentation;
3. A sufficient amount of information so that the experiment can be replicated and built upon; and
4. A community of researchers that understand experimentation, the need for replication, and are willing to collaborate and replicate.

With respect to the Basili/Reiter study introduced in section 1, we can note that while it satisfied criteria 1 and 3, it failed with respect to criteria 2 and 4. It was not suggested by the authors that other researchers might vary the design or manipulate the processes or criteria used for evaluation (although the analysis of the data was varied in a later study [6]). Nor was there a community of researchers willing to analyze the hypotheses even if suggestions for replication had been made.

In contrast, the set of experiments on reading, discussed in a working group at the 1997 annual meeting of

[4] More information is available at the URL http://wwwagse.informatik.uni-kl.de/ISERN/isern.html

ISERN [18], is an example that we have built up a body of knowledge by independent researchers working on different parts of the problem and exposing their conclusions to different plausible rival hypotheses. We have shown in this paper that experimental constraints in software engineering research make it very difficult, and even impossible, to design a perfect single study. In order to rule out the threats to validity, it is more realistic to rely on the "parsimony" concept rather than being frustrated because of trying to completely remove them. This appeal to parsimony is based on the assumption that the evidence for an experimental effect is more credible if that effect can be observed in numerous and independent experiments each with different threats to validity [11].

A second conclusion is that empirical research must be a collaborative activity because of the huge number of problems, variables, and issues to consider. This complexity can be faced with extensive brainstorming, carefully designing complementary studies that provide coverage of the problem and solution space, and reciprocal verification.

It is our contention that interesting and relevant hypotheses can be identified and investigated effectively if empirical work is organized in the form of families of related experiments. In this paper, we have raised several reasons why such families are necessary:

- To investigate the effects of alternative values for important attributes of the experimental models;
- To vary the strategy with which detailed hypotheses are investigated;
- To make up for certain threats to validity that often arise in realistically designed experiments.

Discussion within the experimental community is also needed to address other issues, such as what constitutes an "acceptable" level of confidence in the hypotheses that we address as a community. By running carefully designed replications, we can address threats to validity in specific experiments and accumulate evidence about hypotheses. However, we are unaware of any useful and specific guidelines that concern the amount of evidence that must be accumulated before conclusions can confidently be drawn from a set of related experiments, in spite of the existence of specific threats. More discussion within the empirical software engineering community as to what constitutes a sufficient body of credible knowledge would be of benefit.

Building up a body of knowledge from families of experiments has the following benefits for the software engineering researcher:

- It allows the results of several experiments to be combined in order to build up our knowledge about software processes.
- It increases the effectiveness of individual experiments, which can now contribute to answering more general and abstract hypotheses.
- It offers a framework for building relevant practical software engineering knowledge, organized around the GQM goal template or another framework from the literature.
- It provides a way to develop and integrate laboratory manuals, which can facilitate and encourage the types of replications that are necessary to expand our knowledge of basic principles.
- It helps generate a community of experimenters, who understand the value of, and can carry out, the needed replications.

The ability to carry out families of replications has the following benefits for the software engineering practitioner:

- It offers some relevant practical SE knowledge; fully parameterizing process, product, and context models allows a better understanding of the environment in which the experimental results hold.
- It provides a better basis for making judgements about selecting process, since practitioners can match their development context to the ones under which the processes are evaluated.
- It shows the importance of and ability to tailor "best practices", that is, it shows how software processes can be altered by meaningful manipulation of key variables.
- It provides support for defining and documenting processes, since running related experiments assists in determining the important process variables.
- It allows organizations to integrate their experiences by making explicit the ways in which experiences differ (i.e. what the relevant process, product, and context models are) or are similar, and allowing the

abstraction of basic principles from this information.

Acknowledgements

This work was supported by NSF grant CCR9706151, NASA grant NCC5170, and UMIACS. The authors would like to thank Michael Fredericks and Shari Lawrence Pfleeger for their valuable comments on earlier drafts of this paper.

References

[1] V.R.Basili, "The experimental paradigm in software engineering", Experimental Software Engineering Issues: Critical Assessment and Future Directions, International Workshop, Dagstuhl, Germany, 1992. Appeared in Springer-Verlag, Lecture Notes in Computer Science, Number 706, 1993.

[2] V. R. Basili, "Evolving and packaging reading technologies", *Journal of Systems and Software*, vol. 38, no. 1, pp.3-12, July 1997.

[3] V. Basili, G. Caldiera, F. Lanubile, and F. Shull, "Studies on reading techniques", *Proc. of the Twenty-First Annual Software Engineering Workshop*, SEL-96-002, Goddard Space Flight Center, Greenbelt, Maryland, pp.59-65, December 1996.

[4] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Soerumgaard, M. Zelkowitz, "The empirical investigation of perspective-based reading"; *Empirical Software Engineering Journal*, vol. 1, no. 2, 1996.

[5] V. R. Basili, S. Green , O. Laitenburger, F. Lanubile, F. Shull, S. Sørumgård, and M. Zelkowitz, "Packaging researcher experience to assist replication of experiments", *Proc. of the ISERN meeting 1996*, Sydney, Australia, 1996.

[6] V. R. Basili, and D. H. Hutchens, "An empirical study of a syntactic metric family", *IEEE Transactions on Software Engineering*, vol. SE-9, pp.664-672, November 1983.

[7] V. R. Basili, and R. W. Reiter, "A controlled experiment quantitatively comparing software development approaches", *IEEE Transactions on Software Engineering*, vol. SE-7, no. 3, pp.299-320, May 1981.

[8] V. R. Basili, and H. D. Rombach, "The TAME project: Towards improvement-oriented software environments", *IEEE Transactions on Software Engineering*, vol. SE-14, no. 6, June 1988.

[9] V. R. Basili, R. W. Selby, and D. H. Hutchens, "Experimentation in software engineering", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 7, pp. 733-743, July 1986.

[10] V. Basili, F. Lanubile, F. Shull, "Investigating maintenance processes in a framework-based environment", *Proc. of the Int. Conf. on Software Maintenance*, Bethesda, Maryland, pp.256-264, 1998.

[11] D. T. Campbell, and J. C. Stanley, *Experimental and Quasi-Experimental Designs for Research*, Boston: Houghton Mifflin Co, 1963.

[12] Composable Systems Group, "Model Problems", http://www.cs.cmu.edu/~Compose/html/ModProb/, 1995.

[13] P. Fusaro, F. Lanubile, and G. Visaggio, "A replicated experiment to assess requirements inspections techniques", *Empirical Software Engineering Journal*, vol.2, no.1, pp.39-57, 1997.

[14] IEEE. Software Engineering Standards. IEEE Computer Society Press, 1987.

[15] C. M. Judd, E. R. Smith, and L. H. Kidder, *Research Methods in Social Relations*, sixth edition, Orlando: Harcourt Brace Jovanovich, Inc., 1991.

[16] O. Laitenberger, and J. M. DeBaud, "Perspective-based reading of code documents at Robert Bosch GmbH", *Journal of Information and Software Technology*, 39, pp.781-791, 1997.

[17] F. Lanubile, "Empirical evaluation of software maintenance technologies", *Empirical Software Engineering Journal*, vol.2, no.2, pp.95-106, 1997.

[18] F. Lanubile, "Report on the results of the parallel project meeting reading techniques",

http://seldi2.uniba.it:1025/isern97/readwg/index.htm , October 1997.

[19] F. Lanubile, F. Shull, V. Basili, "Experimenting with error abstraction in requirements documents", *Proc. of the 5th Int. Symposium on Software Metrics*, Bethesda, Maryland, pp.114-121, 1998.

[20] C. M. Lott, and H. D. Rombach, "Repeatable software engineering experiments for comparing defect-detection techniques", *Empirical Software Engineering Journal*, vol.1, no.3, pp.241-277, 1996.

[21] K. Popper, *The Logic of Scientific Discovery*, Harper Torchbooks, New York, NY, 1968.

[22] A. Porter, L. Votta, V. Basili, "Comparing detection methods for software requirements inspections: a replicated experiment", *IEEE Transactions on Software Engineering*, vol. 21, no. 6, pp. 563-575, 1995.

[23] F. Shull, F. Lanubile, and V. R. Basili, "Investigating Reading Techniques for Framework Learning", *Technical Report CS-TR-3896*, UMCP Dept. of Computer Science, *UMIACS-TR-98-26*, UMCP Institute for Advanced Computer Studies, *ISERN-98-16*, International Software Engineering Research Network, May 1998.

[24] F. Shull. *Developing Techniques for Using Software Documents: A Series of Empirical Studies.* Ph.D. thesis, University of Maryland, College Park, December 1998.

[25] F. Shull, "Reading Techniques for Object-Oriented Frameworks", http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/sbr_package/manual.html.

[26] F. Shull, "Lab Package for the Empirical Investigation of Perspective-Based Reading", http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/pbr_package/manual.html.

[27] Z. Zhang, V. Basili, and B. Shneiderman, "An Empirical Study of Perspective-based Usability Inspection", Human Factors and Ergonomics Society Annual Meeting, Chicago, Oct. 1998.

Using Experiments to Build a Body of Knowledge

Victor R. Basili

Experimental Software Engineering Group
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
and
Fraunhofer Center - Maryland

1998

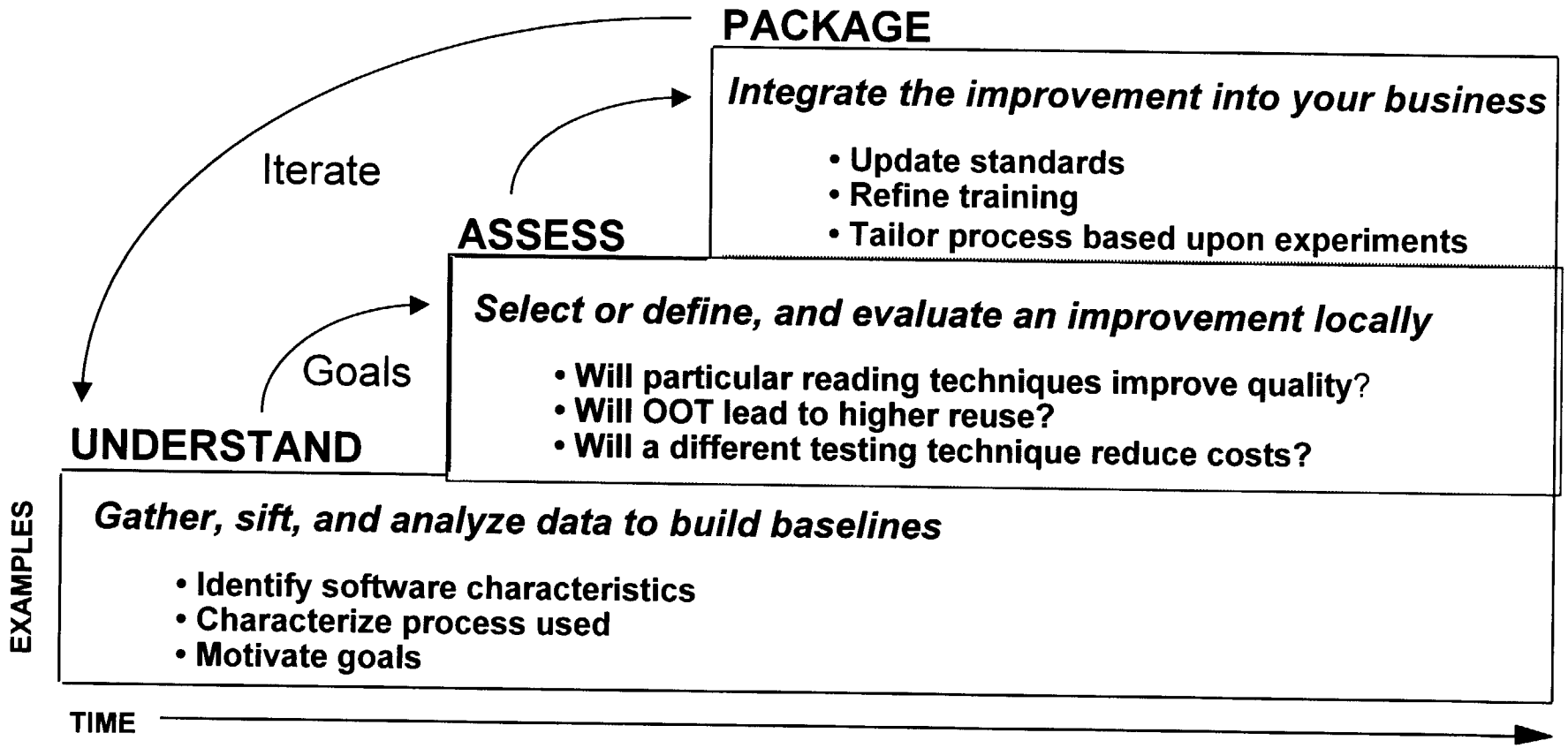
Evolving Knowledge in a Discipline

- Understanding a discipline involves learning, i.e.,
  - observation
  - reflection, and encapsulation of knowledge
  - model building (application domain, problem solving processes)
  - experimentation
  - model evolution over time
- This is the paradigm that has been used in many fields,
  - e.g., physics, medicine, manufacturing.
- The differences among the fields are
  - **how models are built and analyzed**
  - **how experimentation gets done**

Evolving Knowledge
In Software Engineering

- **Software engineering** is a **laboratory science**
- We need to understand the nature of the processes, products and the relationship between the two in the context of the system
- All software is not the same
  - there are a large number of variables that cause differences
  - their effects need to be understood and studied
- Currently,
  - **insufficient set of models** to reason about the discipline
  - **lack of recognition of the limits** of technologies for the context
  - there is **insufficient analysis and experimentation**
- This talk is about experimentation in the software discipline

Where Experiments/Knowledge Building fits in the Quality Improvement Paradigm



Evolving Bodies of Knowledge from Experiments

- Many categories: from controlled experiments to case studies
- Performed for many purposes: to study process effects, product characteristics, environmental constraints (cost or schedule).
- Typically they are looking for a relationship between two variables, such as the relationship between process characteristics and product characteristics
- **Problems** with experiments (controlled)
  - the **large number of variables** that cause differences
  - deal with **low level issues**, **microcosm of reality**, **small set of variables**
- **=> Combining experiments** is necessary to build a body of knowledge that is useful to the discipline

Criteria for building comprehensive bodies of knowledge in Software Engineering

- Sets of **high level hypotheses**
  - address interest of the software engineering community
  - identify sets of dependent and independent variables
  - provide options for the selecting detailed hypotheses
- Sets of **detailed hypotheses**
  - written in a context that allow for a well defined experiment
  - combinable to support high level hypotheses
- **Context variables** that can be changed to allow for
  - experimental design variation (make up for validity threats)
  - specifics of the process context;
- **Sufficient documentation** for replication and combination
- **Community of researchers** willing to collaborate and replicate.

Choosing a High Level Focus

- General Interest to the community
  - **Analyzing the Effects of a SE Process on a Product**

- What are the high level **questions of interest**?
  - Can we effectively design and study techniques that are procedurally defined, document and notation specific, goal driven, and empirically validated for use?
  - Can we demonstrate that a procedural approach to a software engineering task could be more effective than a less procedural one under certain conditions?

- What are the **high level hypotheses**?
  - A reading technique that is procedurally defined, document and notation specific, and goal driven for use is more effective than one that does not have these characteristics
  - A procedural approach to reading based upon specific goals will find different defects than one based upon different goals

Example: Understanding for Use
Motivation for Reading

Why pick reading?

- Reading is a **key technical activity** for analyzing and constructing software documents and products
- Reading is **a model for writing**
- Reading is **critical for reviews, maintenance, reuse, ...**

What is a reading technique?

- a concrete set of instructions given to the reader saying how to read and what to look for in a software product

More Specifically, software reading is

- **the individual analysis of a software artifact**
  - e.g., requirements, design, code, test plans
- **to achieve the understanding needed for a particular task**
  - e.g., defect detection, reuse, maintenance

Choosing a High Level Focus

- How do we build a framework for combining hypotheses from individual experiments, isolating out individual variables?
- Consider using the **Goal/Question/Metrics Paradigm**
- Goal Template:
  - Analyze an **object of study** in order to **purpose** with respect to **focus** from the point of view of **who** in the context of **environment**
- Consider decomposing each of the variables to identify and classify the independent, dependent, and context variables

Choosing a High Level Focus

- Analyzing the Effects of SE Processes on Products
  - Analyze **processes** to evaluate their effectiveness on a product from the point of view of the knowledge builder in the context of (variable set)

- Characterize the object of study:
  - Object of Study (**Process**, Product, ...)
  - Process Class (Life Cycle Model, Method, **Technique**, Tool, ...)
  - Technique Class (**Reading**, Testing, Designing, ...)

- Analyze **reading techniques** to evaluate their effectiveness on a product from the point of view of the knowledge builder in the context of variable set

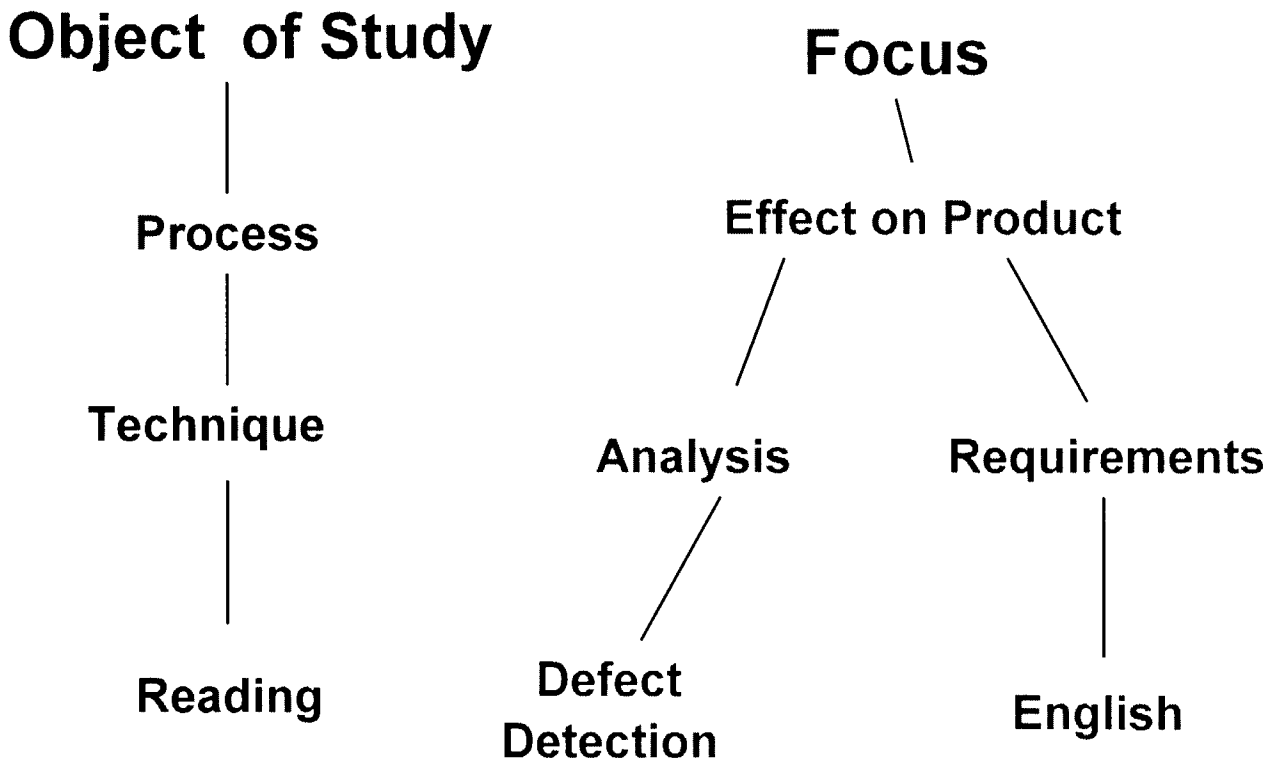
Choosing a High Level Focus

- Analyze reading techniques to evaluate their **effectiveness on products** from the point of view of the knowledge builder in the context of variable set (G1)

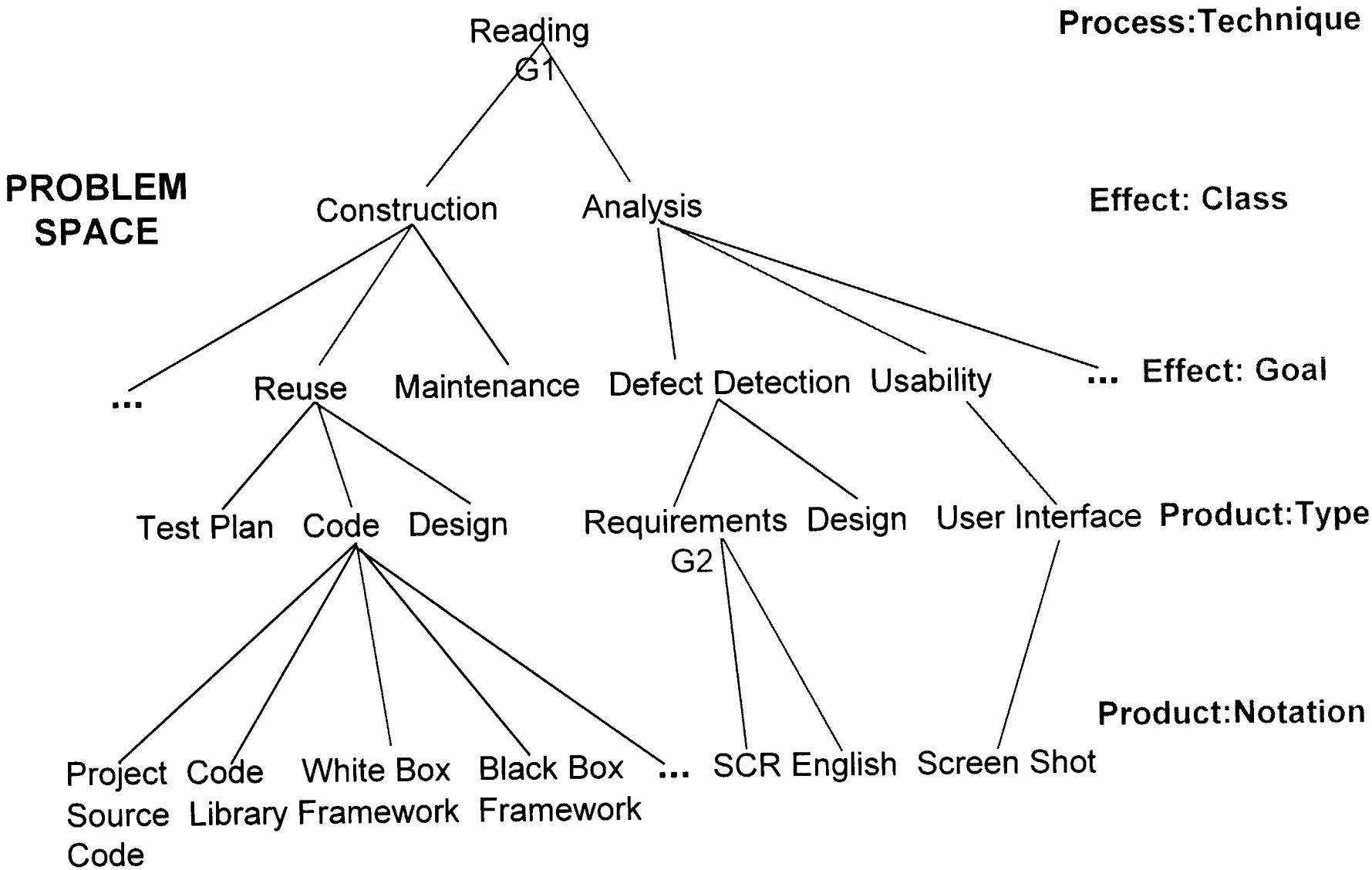
- Characterize the focus: **Effectiveness on a Product**
  - Effectiveness Class (Construction, **Analysis**, ...)
  - Effectiveness Goal (**Defect Detection**, **Usability**, ...
  - Product Type (**Requirements**, Design, Test Plan, **User Interface**, ...
  - Product Notation (**English, SCR**, Mathematics, **Screen Shot**, ...

- Example Goal: Analyze reading techniques to evaluate their **ability to detect defects in a Requirements Document** from the point of view of the knowledge builder in the context of variable set (G2)

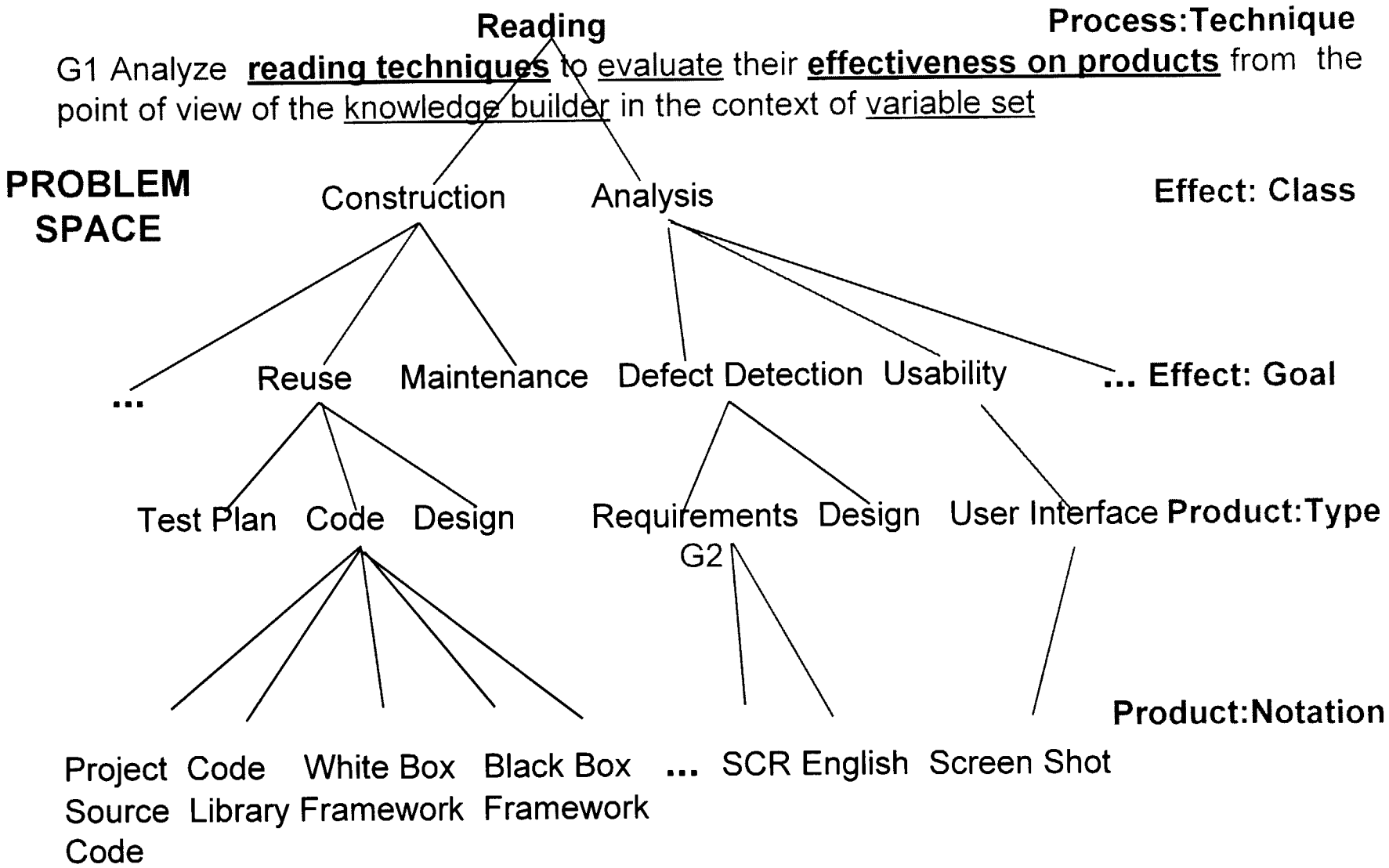
Refining a High Level Focus



Families of Reading Techniques



Families of Reading Techniques



Scenario-Based Reading Definition

- Given this set of characteristics/dimensions, an approach to generating a family of reading techniques, called **operational scenarios**, has been defined

- **Goals:** To define a set of reading technologies that can be
  - document and notation specific
  - tailorable to the project and environment
  - procedurally defined
  - goal driven
  - focused to provide a particular coverage of the document
  - empirically verified to be effective for its use
  - usable in existing methods, such as inspections

- These goals defines a set of guidelines/characteristics for a process definition for reading techniques that can be studied experimentally

Choosing a Specific Focus from the Experimental Framework

- Characterize the process:
  - Technique Class (**Reading**, Testing, Designing, ...)
  - Technique Characteristics (**goal oriented**, **procedurally based**, **coverage focussed**, **documentation and notation specific**, ...)

- Analyze a **set of goal-oriented, procedurally-based, coverage focussed, document and notation specific reading techniques** to evaluate their effectiveness on a product from the point of view of the knowledge builder in the context of (variable set)

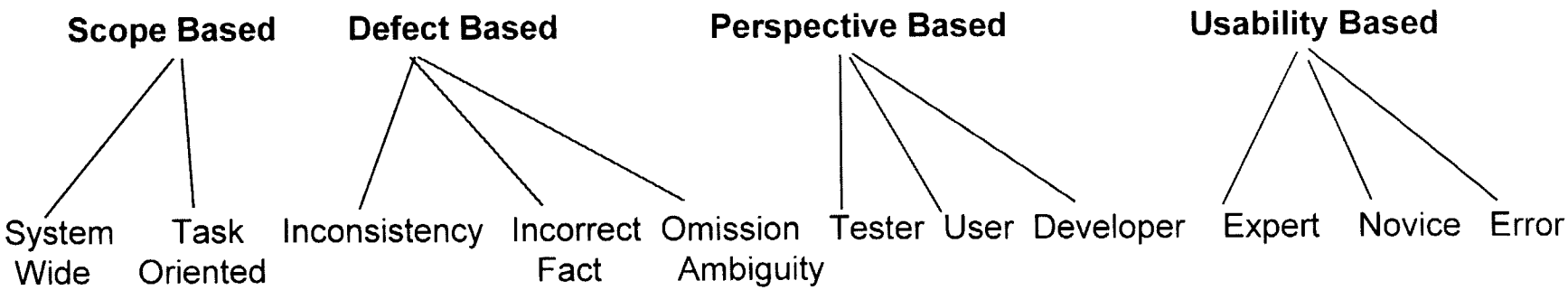
- Analyze a set of scenario based reading techniques to evaluate their effectiveness on products from the point of view of the knowledge builder in the context of (variable set)

- Attempts to satisfy the high level hypotheses and provide a frameworks for individual experiments

Choosing a Specific Focus from the Experimental Framework

- Analyze a set of scenario based reading techniques to evaluate their effectiveness on products from the point of view of the knowledge builder in the context of (variable set)

- We have developed four families of reading techniques
  - parameterized for use in different contexts and
  - evaluated experimentally in those contexts



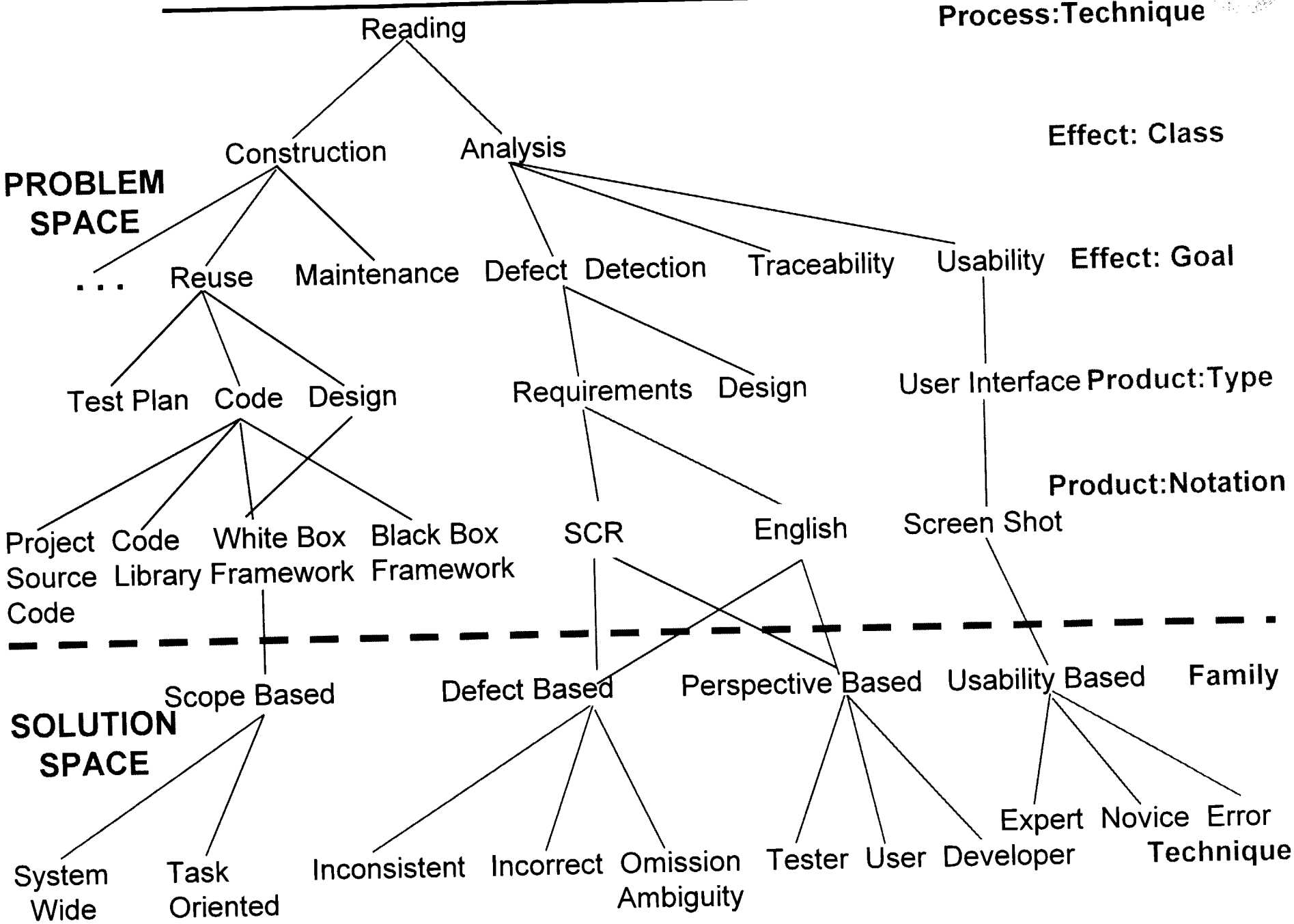
Choosing a Specific Focus from the Experimental Framework

- Analyze a **set of scenario based reading techniques** to evaluate their **ability to detect defects in a Requirements Document** from the point of view of the knowledge builder in the context of (variable set)

- Example: Perspective -Based Reading:
  - Choose perspectives; designer, tester, user
  - Define procedural processes for each perspective
  - Choose experimental treatment
  - Choose defect classes
  - etc.

- Contexts (context variables) can be continually expanded, e.g., NASA/SEL subjects, Professional Software Engineering student, Bosch project personnel

Families of Reading Techniques



Sample Set of Experiments

- We have run several experiments
  - on all four families of reading techniques
  - parameterized for use in different contexts
  - some involved us as directly as experimenters, others did not

- Example Contexts: (Government, University, Industry)
  - NASA/GSFC (PBR)
  - UM Professional SE Course (PBR, UBR)
  - UM Students (DBR, UBR, SBR)
  - Bureau of Census (UBR)
  - Robert Bosch (PBR)
  - Lucent (DBR)

- Example Countries: (U.S., Germany, Italy, Sweden, Scotland, Norway,...)

Choosing a Specific Focus from the Experimental Framework

- There are still many questions that need to be covered:
  - Process variable (**Independent variable**) issues:
    - How do we define/specify the process?
    - How do we account for process conformance?
  - Effectiveness of Product (**Dependent variable**) issues:
    - How do we select good criteria for effectiveness?
  - **Context Variables** Issues:
    - What subjects are performing the process?

- Questions associated with the variables need to be further specified and documented for replication

- Varying the values of these variables allow us to
  - vary the detailed hypotheses
  - support validity of study results

Designing Detailed Experiments to Increase Knowledge

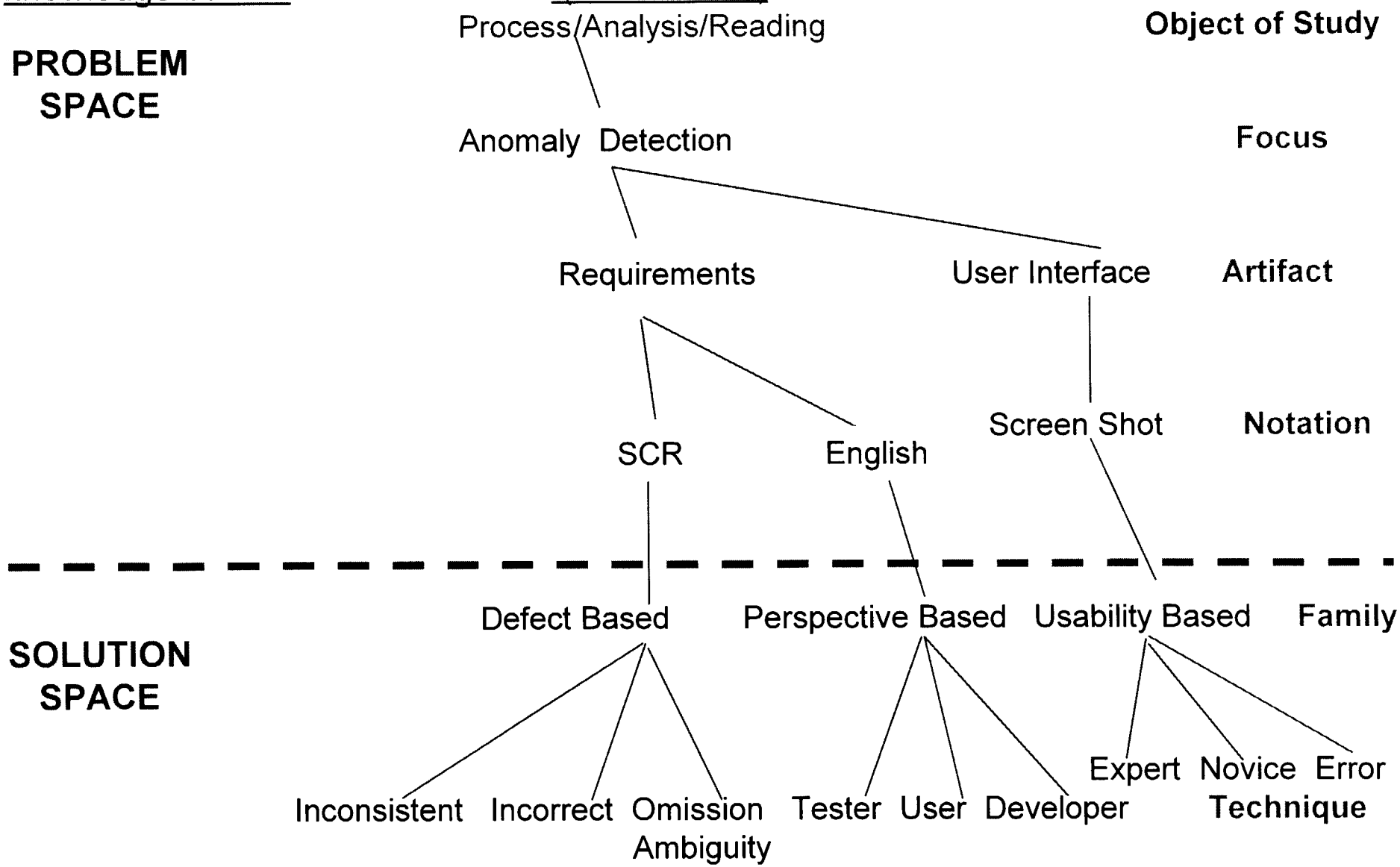
- We can build up knowledge by **replicating** detailed experiments, keeping the same hypothesis, combining results

- Varying **Context Variables**
  - subject experience
  - context (classroom, toy, off-line, in project)
  - variability among subjects
  - Vary order of events and activities

- Allows us to balance threats to validity
  - interaction of experience and treatment
  - spontaneous migration of subjects across treatments
  - replicating to counterbalance

Focused Families of Analysis Techniques

G3 Analyze a **set of processes focused to provide a particular coverage of an artifact** to evaluate their **ability to detect anomalies** from the point of view of the knowledge builder in the context of (variable set)



Conclusions from Experiments

- Able to **combine** the **results** of several experiments and **build up** our **knowledge** about software processes
  - We can **effectively design and study techniques** that are procedurally defined, document and notation specific, goal driven, and empirically validated for use
  - We can demonstrate that a **procedural approach** to a software engineering task could be more effective than a less procedural one under certain conditions (e.g., depends on experience)
  - A procedural approach to reading based upon **specific goals** will find defects related to those goals, so reading can tailored to the environment
  - et. al.

Conclusions about Knowledge Building Experimental Framework

- Benefit to Researchers:
  - ability to **increase the effectiveness** of individual experiments
  - offers a **framework** for building relevant practical SE knowledge
  - provides a way to develop and integrate **laboratory manuals**
  - generate a **community** of experimenters

- Benefits to Practitioners:
  - offers some relevant **practical SE knowledge**
  - provides a better basis for making judgements about **selecting process**
  - shows importance of and ability to tailor "**best practices**"
  - provides support for defining and **documenting processes**
  - allows organizations to **integrate their experiences** with processes

Contributors to This Work

- Directly to the Ideas Presented here:
  - Forrest Shull, Filippo Lanubile
- As Experimenters Locally:
  - Reported Experiments: Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shull, Sivert Sorumgaard, Marvin Zelkowitz, Zhijun Zhang
  - New Studies Underway: Fred Fredericks, Shari Lawrence Pfleeger, Rae Kwon, Guilherme Travassos
- As Experimenters in Other Locations
  - ISERN members
  - Others

Session 2: Experimentation

Culture Conflicts in Software Engineering Technology Transfer
D. Wallace, National Institute Of Standards and Technology,
and M. Zelkowitz, University Of Maryland

An Adaptation of Experimental Design to Empirical Validation of Software Engineering Theories
N. Juristo and A. Moreno, Universidad Politecnica de Madrid

Disciplined Software Engineering: Extending Enterprise Engineering Architectures to Support the OO Paradigm
F. Maymir-Ducharme, Lockheed Martin

54-61

Culture Conflicts in Software Engineering Technology Transfer

Marvin V. Zelkowitz*
Department of Computer Science and
Inst. for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
and Fraunhofer Center - Maryland
College Park, Maryland 20742

Dolores R. Wallace
Information Technology Laboratory
Natl. Inst. of Standards and Technology
Gaithersburg, Maryland 20899

David W. Binkley
Computer Science Department
Loyola College
Baltimore, Maryland
and Information Technology Lab.
Natl. Inst. of Standards and Technology
Gaithersburg, MD 20899

Abstract

Although the need to transition new technology to improve the process of developing quality software products is well understood, the computer software industry has done a poor job of carrying out that need. All too often new software technology is touted as the next "silver bullet" to be adopted, only to fail and disappear within a very short period. New technologies are often adopted without any convincing evidence that they will be effective, yet other technologies are ignored despite the published data that they will be useful. Clearly there is a clash between those developing new technologies and those responsible for developing quality products. In this paper we discuss a study conducted among a large group of computer software professionals in order to understand what techniques can be used to support the introduction of new technologies, and to understand the biases and opinions of those charged with researching, developing or implementing those new technologies. This study indicates which evaluation techniques are viewed as most successful under various conditions. We show that the research and industrial communities do indeed have different perspectives, which leads to a clash between the goals of the technology researchers and the needs of the technology users.

Keywords: Experimentation, Survey, Technology transfer, Validation models

1. Introduction

When the computer industry began several decades ago, software engineering was somewhat unique among engineering fields in that researchers and practitioners worked closely together in using and understanding this new technology. There was easy cross-fertilization between these two communities. Over time, this has changed with tremendous growth of computer applications, computer users, and computing professionals. Programming languages have evolved from-low level assembler languages to today's very high level visual object-oriented languages. Simple programs have become complex large systems, with some systems running an entire enterprise. Methods for developing programs have grown from design-writing on napkins to a myriad of overlapping processes comprising varieties of methods and documentation types.

A response to this growth has been a corresponding growth in organizations dedicated to supplying an ever-increasing need for better tools and techniques for producing these complex products. Trade shows, research conferences, trade magazines proliferate on the technology scene. New professional technical

* Research supported in part by National Science Foundation grant CCR-9706151 to the University of Maryland.

journals regularly come alive to add to an already large number; the IEEE alone through its Computer Society currently publishes 20 monthly or bimonthly computer technology publications.

In spite of an abundance of methods and tools and information about them, why do the same problems appear over and over again in new software developments? Why are development schedules not met? Why do some systems fail? Why do some technical problems remain unsolved? While new solutions are frequently proposed, many have not been transferred into the industry at large. Many problems remain untouched by researchers. Why does it appear that today researchers and practitioners are no longer necessarily understanding each other's needs and efforts?

Researchers have been looking at the role of experimentation in computer science research [Fenton94]. However, most of these have looked at the relatively narrow scope of how to conduct replicated scientific experiments within this domain. We have been looking at the larger problems of the role of experimentation as an agent in transferring new technology into industry. We have been studying various experimental methods, in addition to the replicated experiment, useful for validating newly developed software technology [Zelkowitz97] [Zelkowitz98], and we have also studied various evaluation methods industry uses before adopting a new technology. As we later explain, these two processes are very different. The questions important to us include "Which of these validation and evaluation methods are most effective?" "Why aren't these methods used more often?" and "Why don't these results provide evidence for the transference of a technology into industry?" To try to understand these questions, we decided to survey a cross section of computer professionals about their views about software engineering technology validation.

1.1 The research and industrial communities

Researchers, whether in academia or industry, have a desire to develop new concepts and are rewarded when they produce new designs, algorithms, theorems, and models. The "work product" in this case is often a published paper demonstrating the value of their new technology. Development professionals, however, have a desire and are paid to produce a product using whatever technology seems appropriate for the problem at hand. The end result is a product that produces revenue for their employer.

Researchers select their research according to a topic of their own interest; the topic may or may not be directly related to a specific problem faced by industry. After achieving a result that they consider interesting, they have a great desire to get that result in print. Providing a good scientific validation of the technology is often not necessary for publication, and several studies have shown that experimental validation of computer technology is particularly weak, e.g., [Tichy95] [Zelkowitz98].

In industry, producing a product is most important and the "elegance" of the process used to produce that product is less important than achieving a quality product on time as a result. Being "state of the art" in industry often means doing things as well (or as poorly) as the competition, so there is considerable risk aversion to try a new technology unless the competition is also using it.

Consequently, researchers produce papers outlining the values of new technology, yet industry often ignores that advice. Assorted "silver bullets" are proposed as solutions to the "software crisis" without any good justification that they may be effective, are used for a time by large segments of the community, and then are discarded when they indeed turn out not to be *the* solution. Clearly the research community is not generating results that are in tune with what industry needs to hear, and industry is making decisions without the benefit of good scientific developments. The two communities are severely out of touch with

one another. The purpose of our survey is to try and understand these communities and understand their differences.

1.2 Research models

We began our effort to understand the differences between the research and industrial communities by examining models of experimentation for computer technology research. We identified 12 methods of experimentation that have been used in the computer field [Table 1.1] and verified their usage by studying 612 papers appearing in three professional publications at 5-year intervals [Zelkowitz98] from 1985 through 1995. About 20% of the papers contained no validation at all and another third contained only a weak ineffective form of validation. The figure for other scientific fields was more like 10% - 15% [Zelkowitz97]. The methods are defined in Appendix 1.

Table 1.1 Experimental Validation Models	
Case study	Project monitoring
Dynamic analysis	Replicated
Field study	Simulation
Legacy data	Static analysis
Lessons learned	Synthetic
Literature search	Theoretical analysis

Our results were consistent with those found by Tichy in his 1995 study of 400 research papers [Tichy95]. He found that over 50% of the design papers did not have any validation in them. In a more recent paper [Tichy98], Tichy makes a strong argument that more experimentation is needed and refutes several myths deprecating the value of experimentation.

1.3 Transition models

Given the set of research validation methods, we then sought to determine the techniques actually used by industry in order to transition a new technology. We visited several large development corporations[1] and interviewed reasonably high level individuals, such as Chief Scientist, Chief Technology Officer, and managers of large divisions. All had ultimate responsibility for technology selection. They were primarily influenced by trade shows, weekly trade magazines, Web information, customer opinion (i.e., technologies that would win the contract), vendor opinion, friends in other companies, and infrequently by the papers in professional technical journals. Sometimes recommendations from technical staff would be based on their readings and would eventually reach the managers' offices. Once a technology was identified, the companies might perform a pilot study or were mentored by an expert of the technology to determine if the technology would be effective.

Based on these industrial interviews and some earlier work by Brown and Wallnau [Brown96], we defined a set of industrial transition models for technology evaluation. While the transition models include some that are similar to those of the researchers, many are different [Table 1.2]; Appendix 2 provides a short description of these models. For example, vendor opinion (e.g., trade shows, weekly trade magazines, web information) seemed important to industry; Web information also provides access to research literature so we needed to separate the medium in which information is located from the type of model that information supports. An important finding, though, is that everyone with whom we spoke claimed to use the web to find technology information.

[1] To assure frank discussion, we agreed not to reveal the names of the corporations who spoke with us.

Table 1.2 Industrial Transition Models	
Case study	Research literature
Data mining	Shadow (replicated) project
Demonstrator projects	State of the art
Feature benchmark	Survey
Field study	Theoretical analysis
Measurement	Vendor opinion
Pilot study	

Our interviews revealed that a company may use people-oriented methods for technology transfer. For example, a company may hire a well-recognized expert in that technology, perhaps its creator, to help integrate the method into company practices. They may specifically recruit people who have that skill on their resumes. Another practice appears to be training by hiring an expert to teach in-house training or by sending their personnel to universities or training companies.

In retrospect we would have entered these models in our survey, especially because the survey results discussed in Section 4 indicate that in two instances, two models could have been combined. Field study and survey both estimate the probable effects of some new technology. In the field study, several development groups may be observed over a short time period while in the survey several experts may discuss their opinions based on their expertise in the technology. They are rather closely aligned in time and people requirements and were perceived approximately the same. A pilot study involves a sample project, usually small, to study a new technique while demonstrator studies are less complete multiple instances of a pilot study.

1.4 Understanding each community

Researchers principally use methods from Table 1.1 in order to demonstrate the value of their technological improvements and industry selects new technology to employ by using the methods in Table 1.2. How do these communities interact? How can their methods support forward growth in computer technology and its application in real systems? We need to develop a better understanding of what each community understands and values. Then, perhaps, we can identify commonalities and gaps, and from there, mechanisms to enable each community to benefit better from the other.

2. Development of the survey

To understand the different perceptions between those who develop technology and those who use technology, we decided to survey the software development community to learn how they view the effectiveness of the various evaluation models of Tables 1.1 and 1.2. For questions, we based our survey on a previous survey [Daly97], modified for our current purposes. Each survey participant was to rank the difficulty of each of our 12 experimental models (or 13 evaluation models) according to 7 criteria, criteria 1 and 2 being new and 3 through 7 being the same as the Daly criteria. We decided to try to obtain an objective score by having all values ranked between 1 and 20, with 10 being arbitrarily defined as the maximum difficulty that a given company would apply in practice, and 20 being defined as an impossible model for that criterion.

2.1 Survey questions

The 7 questions we chose were:

1. *How easy is it to use this method in practice?* -- Can we use this method to evaluate a new technology? The answer should be independent of whether the method gives accurate results or not.

2. *What is the cost of adding one extra subject to the study?* -- Assume you want to add an additional subject (another data point) to your sample. What is the relative cost of doing so?

3. *What is the internal validity of the method?* -- What is the extent to which one can draw correct causal conclusions from the study? That is, to what extent can the observed results be shown to be caused by the manipulated dependent experimental variables and not by some other unobserved factor?

4. *What is the external validity of the method?* -- What is the extent to which the results of the research can be generalized to the population under study and to other settings (e.g., professional programmers, organizations, real projects)?

5. *What is the ease of replication?* -- What is the ease with which the same experimental conditions can be replicated (internally or externally) in subsequent studies? It is assumed that the variables that can be controlled (i.e., the dependent variables) are to be given the same value.

6. *What is the potential for theory generation?* -- What is the potential of the study to lead to unanticipated a priori and new causal theories explaining a phenomenon? For example, exploratory studies tend to have a high potential for theory generation.

7. *What is the potential for theory confirmation?* -- What is the potential of the study to test an a priori well-defined theory and provide strong evidence to support it?

In an eighth question we asked each participant to rank the relative importance (again using the 1-20 ranking) of each of the 7 questions when making a decision on using a new technology. That is, which of the 7 questions was most important when a new technology was being evaluated?

We developed two different survey instruments from these 8 questions -- one by ranking each of the 12 research validation methods of Table 1.1 (i.e., the research survey) and one by ranking each of the 13 evaluation methods of Table 1.2 (i.e., the industrial survey).

2.2 Population samples

For our 2 survey instruments we obtained three populations to sample. Sample 1 included U.S.-based authors with email addresses published in several recent software engineering conference proceedings[2]. These were mostly research professionals, although included a few developers. Approximately 150 invitations to participate were sent to these individuals, and 45 accepted. The survey was not sent until the participant agreed to fill out the form, which we estimated would take about an hour to 90 minutes to read and fill out. About half of the individuals returned the completed form.

Sample 2 included U.S.-based authors with email addresses from several recent industry-oriented conferences. They were sent the industrial survey. About 150 invitations to participate were sent and about 50 responded favorably to our invitation. They were then sent the survey. Again, about half completed and returned the form.

[2] The survey was conducted via email.

Sample 3 were students in a graduate software engineering course at the University of Maryland taught by one of the authors of this paper. This sample was given the research survey. This course was part of a masters degree program in software engineering, and almost all of the students were working professionals with experience ranging up to 24 years. Not surprisingly, the return rate of the form for this sample was high at 96% (44 of 46).

It is important to realize that we wanted the subjective opinion of those surveyed on the value of the respective validation techniques based upon several criteria. Not everyone returning the survey had previously used all, or even any, of the listed methods. We simply wanted their views on how important they thought the methods were. However, by choosing our sample populations from those writing papers for conferences or taking courses for career advancement, we believe we have chosen sample populations that are more knowledgeable, in general, about validation methods than the average software development professional. The invitations were sent early in 1998, and data was collected February through early April, 1998. Table 2.1 summarizes the 3 sample populations.

Table 2.1 Characteristics of each survey sample

Sample	Survey	Sample size	Years exper.	Academic Position	Industrial R&D	Industrial developer	Other (e.g., Consultants)
1 (Research)	Research	18	18.6	9	3	3	3
2 (Industry)	Industry	25	19.1	0	5	8	12
3 (Students)	Research	44	6.6	1	5	27	11

3 Survey results

Our initial concern was to determine bias in the set of responses. Would certain individuals rank all techniques high or low compared to other individuals? In order to test for this, we computed the average raw scores for each technique for each question, and we also ranked each answer (i.e., computing the easiest technique for each question, second easiest, third easiest, ..., 12th easiest). This would eliminate such bias, but would also eliminate the significance of the value 10 being the subjective value of "hard to do." Fortunately, we believe that we don't have to take this into account. Figure 1 shows the value for the question "Easy to do." The first column represents the average raw scores for the 12 methods of Table 1.1 from the research sample (sample 1) and the second column is the average ranked score. Low values indicate the more important techniques. The fact that the ordering of the techniques from best to worst was essentially the same indicates that the raw score is an accurate reflection of the ranking. Only the 3rd and 4th, 5th and 6th, and 9th and 10th techniques switched places, not a major change. Columns 3 and 4 represent similar data from the student sample (sample 3). Here only the third and fourth and eighth and ninth techniques switched places. However, there are some slight differences between sample 1 and sample 3, which will be discussed in Section 4.

Similar charts were obtained from the other questions. In addition, the correlation between the raw scores and the ranked scores for sample 1 was 0.86, 0 .96 for sample 2 and 0.93 for sample 3. On this basis, we decided we could use the raw data and did not need to use only the ranked data for comparisons.

The average value for each technique for each of the 7 criteria appears in Figures 2 through 4. Figure 2 represents the average score for each of the 12 experimental methods over all 7 criteria for sample 1 with alpha=.05 confidence interval bars surrounding each average value. The "7" in each criterion represents the midpoint among the methods in order to make it easier to read the figure. Of greatest interest are bars that do not overlap, meaning there is a 95% probability that the average values for those techniques

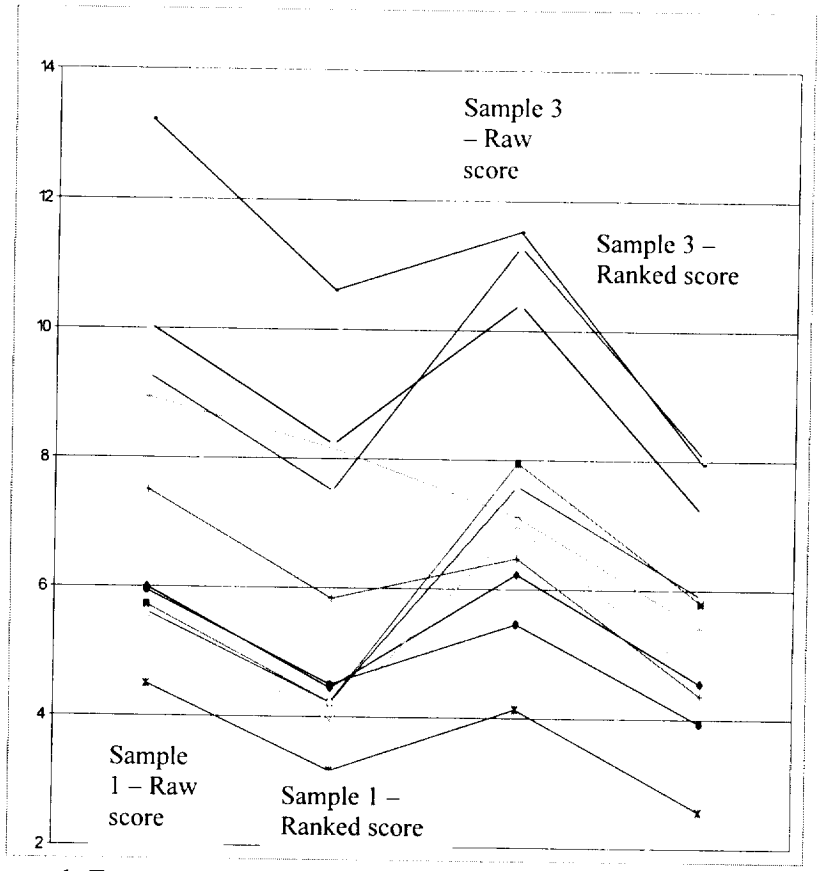


Figure 1. Easy to do. Average value for each of 12 validation methods.

indicate a significant difference. Figure 3 represents a similar graph for sample 2 (the industrial group ranking 13 techniques) and Figure 4 represents a similar graph for sample 3 (the student industrial sample).

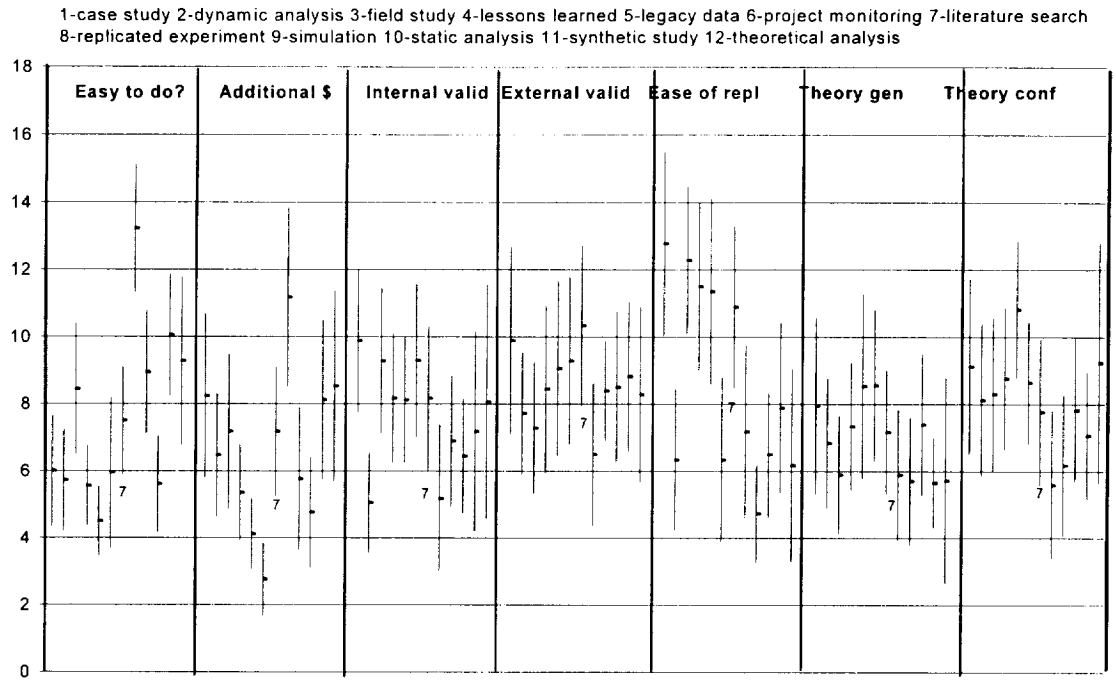


Figure 2. Sample 1 (research group) results.

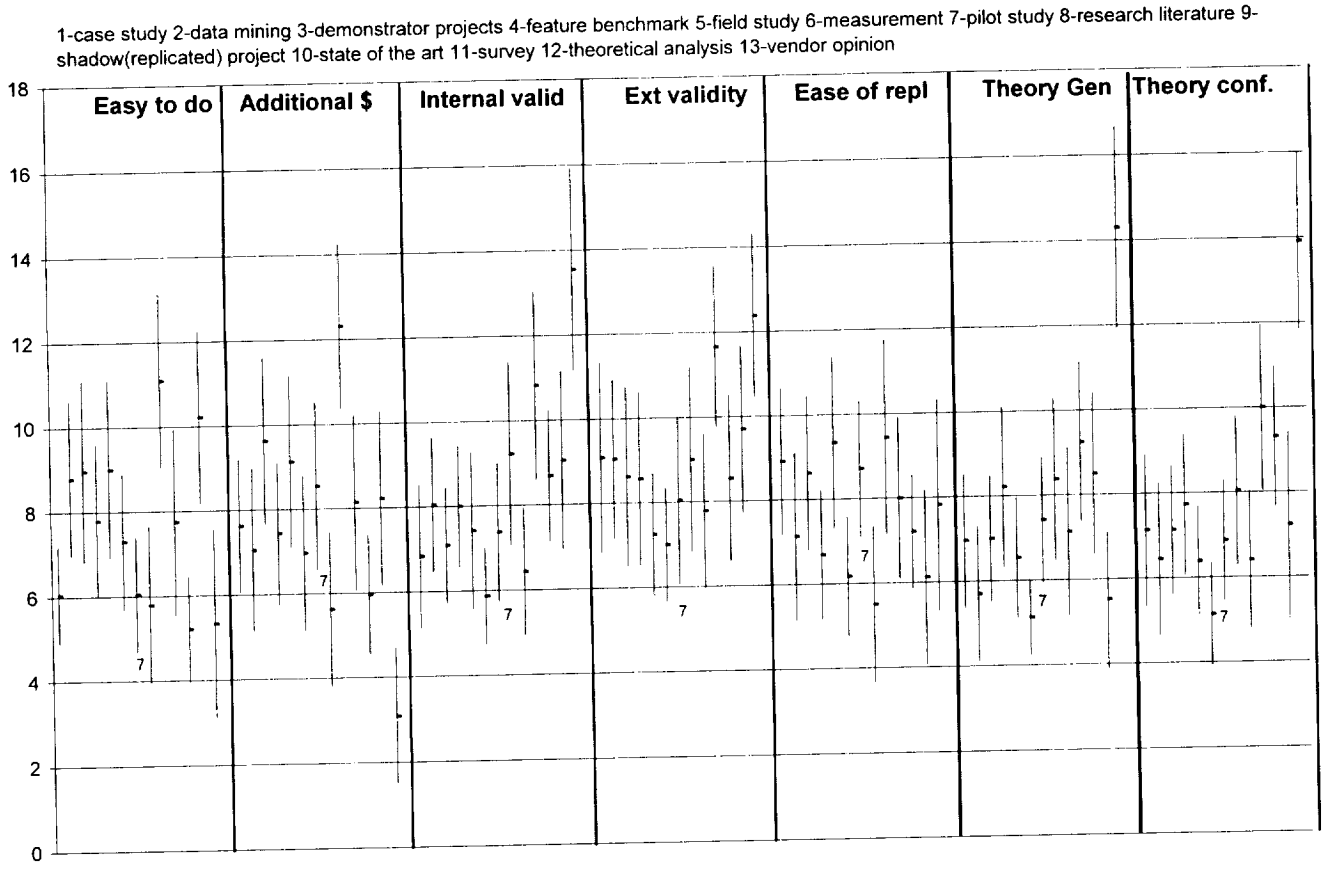


Figure 3. Sample 2 (industry group) results.

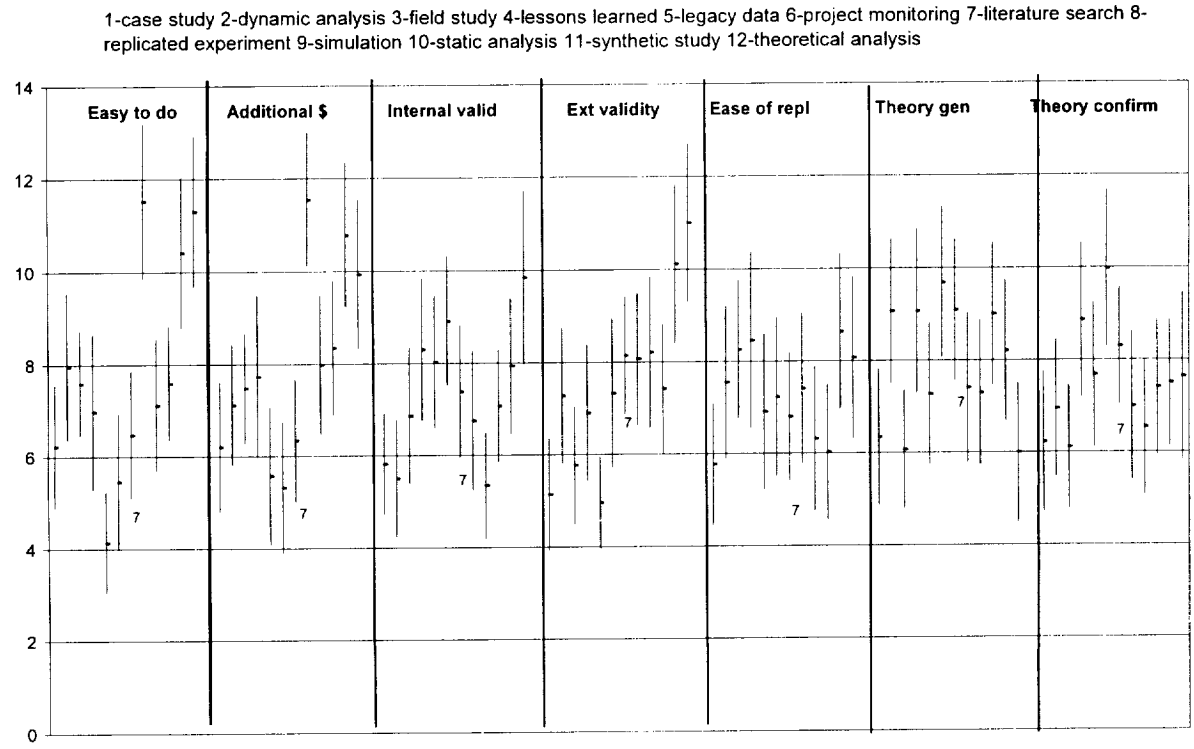


Figure 4. Sample 3 (student industrial group) results.

One way to simplify the data from these figures is to split the methods for each criterion into three partitions: practical, neutral, and impractical. The following procedure was applied:

1. Each method whose upper confidence interval was below the average value for all techniques would be listed in the practical partition. These methods are all "better than average" according to our 95% confidence criterion.
2. Each method whose lower confidence interval was above the average value for all methods would be listed in the impractical partition. These methods are all "worse than average" according to our 95% confidence criterion.
3. All other methods would be listed in the neutral partition.

Tables 3.1 through 3.3 summarize this process giving the practical and impractical techniques. All other methods are in the neutral partition.

Table 3.1 Practical and impractical techniques from research sample

	Easy	Addit. $	Int. val.	Ext. val.	Ease of repl.	Theory gen.	Theory conf.
Practical	Dyn. anal Les. learned Legacy data Static anal.	Legacy data Proj. mon. Static anal.	Dyn. anal. Replication		Dyn. anal. Simulation Static anal.		Replicated
Impractical	Replicated Synthetic	Replicated	Case study		Case study Field study Les. learned		Legacy data

Table 3.2 Practical and impractical techniques from industry sample							
	Easy	**Addit. $**	**Int. val.**	**Ext. val.**	**Ease repl.**	**Theory gen.**	**Theory conf.**
Practical	Case study Pilot study Survey Vendor opin.	Res. Lit Survey Vendor opin.	Measure	Field study Measure	Measure Res. Lit.	Data mining Measure Theory anal.	Field study Measure
Impractical	Replicated	Replicated	State of art Vendor opin	State of art Vendor opin		Vendor opin.	State of art Vendor opin

Table 3.3 Practical and impractical techniques from student industrial sample							
	Easy	**Addit. $**	**Int. val.**	**Ext. val.**	**Ease repl.**	**Theory gen.**	**Theory conf.**
Practical	Case study Legacy data Proj. mon.	Case study Legacy data Proj. mon. Lit. search	Case study Dyn. Anal. Simulation	Case study Legacy data	Case study	Case study Field study Theory anal.	Field study
Impractical	Replicated Synthetic Theory anal.	Replication Synthetic Theory anal.	Proj. mon. Theory anal.	Synthetic Theory anal.		Proj. mon.	Proj. mon.

Our final 8th question was to rate the importance of each of the 7 questions when making a decision on using a new technology. The purpose was to determine which of the criteria was most important when making such a decision. Figure 5 summarizes those answers on a single chart, the column labeled 1 representing the average values for the first sample, column 2 representing the average value for sample 2 and column 3 being sample 3.

4 Survey Evaluation

4.1 Preferred research techniques

Figures 2 and 4 and Tables 3.1 and 3.3 present a summary of our findings for the research validation methods. We summarize some of the observations from those figures.

In terms of easiness (question 1), replicated experiments and synthetic experiments for the research sample and replicated experiments, synthetic experiments and theoretical analysis for the student industrial sample were viewed as significantly (at the .05 level) harder to do than the other techniques and as impractical according to Tables 3.1 and 3.3. With average scores above 10, the consensus of these groups was that industry would never use such techniques as part of a validation strategy. It is no wonder that such techniques are rarely reported in the literature. In our earlier survey [Zelkowitz98] only 3.2% of the reported studies used synthetic or replicated experiments.

On the other hand, these two groups differed in their belief in the effectiveness of theoretical analysis with respect to internal and external validity (questions 3 and 4). Whereas the research group considered a theoretical validation likely to be used as much as any other technique (i.e., in the neutral partition of Table 3.1), the industrial group considered it most difficult to use, preferring instead the "hands on" techniques over the more formal arguments.

Other than the cost and ease issues, none of the other criteria exhibited significant differences among the respondents. However, when we combine the criteria into a single composite number, differences do become apparent (See Section 4.3).

4.2 Preferred industrial methods

Figure 3 and table 3.2 give the basic results for the industrial transition methods. As with the research population, the replicated (shadow) project had an average rating (over all 7 questions) of over 10, signifying little industrial interest in performing such studies. Vendor opinion also averaged above 10, as did the need to be state of the art.

These high scores were all probably due to different reasons. Replicated experiments were viewed as hardest to do (highest score among all techniques at about 13.5), while vendor opinion had the worst internal and external validity (the ability for the method to explain the phenomenon under study, i.e., trusting the vendor to give the correct explanation). On the other hand, the need to be state of the art also suffered with respect to internal and external validity.

It is interesting to note that according to table 3.2, vendor opinion was considered practical according to ease of use (criterion 1), yet was impractical according to the criteria that dealt with accuracy of the evaluation (questions 3, 4, 6 and 7).

Theoretical analysis was harder to do than any other technique except the replicated project.

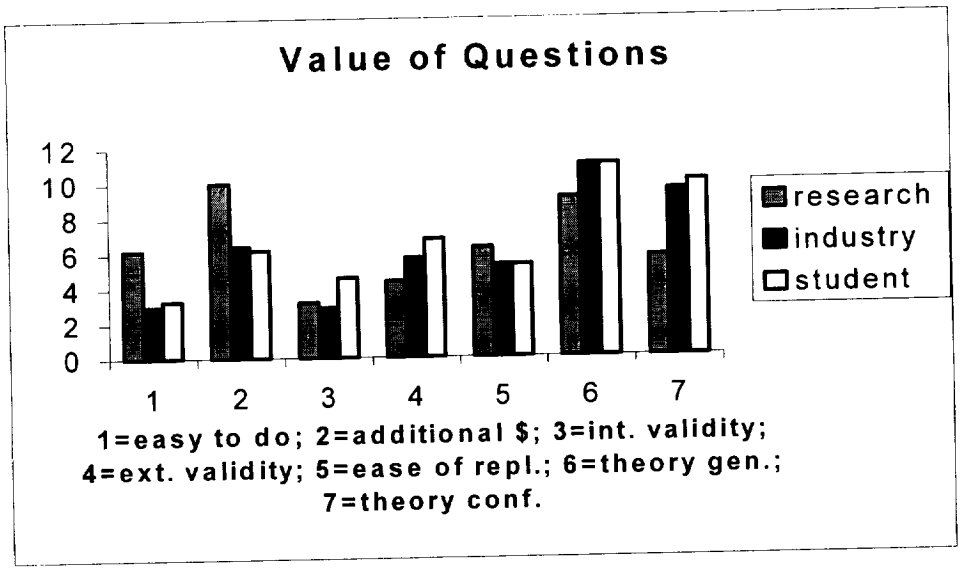


Figure 5. Relative importance of each criterion.

4.3 Culture differences

By comparing results across different samples, we gain an appreciation of the differing values in the software engineering community. Although sample 2 evaluated the industrial methods according to our 7 criteria and sample 3 evaluated the research methods for the same criteria, both were made up mostly of

professional developers. Question 8, the importance of each criterion, reveals strong agreement between these two populations, and strong disagreements with the research professionals from sample 1.

Figure 5 summarizes this result. Both samples 2 and 3 viewed easy to do, internal validity (that the validation confirmed the effectiveness of the technique) and the ease of replicating the experiment as the most important criteria in choosing a new method. While internal validity was important, external validity was of less crucial concern. That can be interpreted as the self-interest of industry in choosing methods applicable to its own environment and of less concern if it also aided a competitor.

On the other hand, for the research community of sample 1, internal and external validity, the ability of the validation to demonstrate effectiveness of the technique in the experimental sample and also to be able to generalize to other samples, were the primary criteria. Confirming a theory was next, obviously influenced by the research community's orientation in developing new theoretical foundations for technology. At the other end of the scale, cost was of less concern where ease of replication was only 5th most important and cost of adding additional subjects was rated as last.

This points out some of the problems we addressed at the beginning of this paper. The research community is more concerned with theory confirmation and validity of the experiment and less concerned about costs, whereas the industrial community is more concerned about costs and applicability in their own environment and less concerned about general scientific results which can aid the community at large.

4.4 Composite measures

Given the set of 7 criteria, can we generate any composite measure for evaluating the effectiveness of the various validation methods? Since we have the respondents' impressions of the importance of each of the 7 criteria (via Figure 5), one obvious composite measure is the weighted sum of all the criteria evaluations. In this case, low score would determine the most significant methods. Table 4.1 gives these results.

Table 4.1 Composite measures

Sample 1 ordering (Research group)		**Sample 3 ordering (Student group)**		**Sample 2 ordering (Industry group)**	
Simulation	288	Case study	284	Measurement	258
Static analysis	292	Legacy data	314	Data mining	305
Dynamic analysis	298	Field study	315	Theoretical analysis	324
Project monitoring	301	Simulation	333	Research literature	325
Lessons learned	339	Dynamic analysis	355	Case study	326
Legacy data	345	Static analysis	361	Field study	327
Synthetic study	346	Literature search	370	Pilot study	329
Theoretical analysis	348	Replicated experiment	387	Feature benchmark	338
Field study	363	Project monitoring	388	Survey	343
Literature search	367	Lessons learned	391	Demonstrator project	345
Replicated experiment	368	Theoretical analysis	405	Replicated project	361
Case study	398	Synthetic study	418	State of the art	407
				Vendor opinion	469

Table 4.1 reveals some interesting observations:

1. For the research community, tools-based techniques dominate the rankings. Simulation, static analysis, and dynamic analysis are techniques that are easy to automate and can be handled in the laboratory. On the other hand, techniques that are labor intensive and require interacting with industrial groups (e.g., replicated experiment and case study) are at the bottom of the list. From our own anecdotal experiences over the past 20 years, working with industry on real projects certainly is harder to manage than building evaluation tools in the lab.
2. For the industrial community (the student sample 3 population), almost the opposite seems true. Those techniques that can confirm a technique in the field using industry data (e.g., case study, legacy data, field study) dominate the rankings, while "artificial" environments (e.g., theoretical analysis, synthetic study) are at the bottom. Again, this seems to support the concept that industrial professionals are more concerned with effectiveness of the techniques in live situations than simply validating a concept.
3. The industrial group evaluating the industrial validation methods (sample 2) cannot be compared with the above two groups since the methods they evaluated were different; however, there are some interesting observations. For one, measurement, the continual collection of data on development practices, clearly dominates the ranking. This is a surprising considering the difficulty the software engineering measurement community has been having in getting industry to recognize the need to measure development practices. With models like the Software Engineering Institute's Capability Maturity Model (CMM), the SEI's Personal Software Process (PSP) and Basili's Experience Factory promoting measurement, perhaps the word is finally getting out about the need to measure. But actual practice does not seem to agree with the desires of the professionals in the field. In addition, theoretical analysis came out fairly high in this composite score, but that does not seem to relate to experiences in the field.
4. Also within the industrial group, the need to be state of the art came near the bottom of the list (12^{th} out of 13) as not important. Basing decisions on vendor opinions was last. Yet vendors often influence the decision making process. Vendor opinions were judged to be least effective with respect to internal and external validity (Figure 3), but since vendor opinion was also judged to be one of the easiest to do, apparently users rely on such opinions even though they know the results are not to be trusted.
5. Data mining of collected data turned out to be second most important according to the industrial group. This is compatible with measurement being most important. If data is not collected, then there is nothing available to mine. Theoretical validation, literature search, and various experimental developments (i.e., field study, case study, pilot study) all ranked about the same level of importance to this group.

5. Conclusions

In this paper we discuss a survey taken from approximately 90 software engineering professionals. The survey evaluated subjective opinions on the value of validation methods for transferring new technology into industry. The idea was to study those methods used by the research community to validate new technologies and those methods used by industry to evaluate a new technology and to try and understand the differences. From this survey, we can make the observation that the research community and the development community do indeed have different perceptions of the role of experimentation to validating new technology. Researchers are more interested in how well a theory has been validated, whereas industry is more attuned, as expected, to how well the technique works in their own environment. Costs, while important to the industry sample, are mostly ignored by the research community.

Publication of research results is a major focus of the research community. In this respect, journal editors can play an important role in affecting this cultural difference. Developing new technologies and getting

them into use should be a major focus of software engineering research. Editors of journals consider requiring more real-world validation using models like case studies, legacy data and field studies and be more suspect at validation via laboratory models, such as simulation and synthetic studies.

The survey also indicates that one should not simply be state of the art simply to be "fashionable" or listen to vendors for technology transfer decisions. Such decisions should depend on more technological reasons. Yet such actions are taken daily.

Measurement became the most important industrial decision making process in our composite analysis, yet anecdotal evidence indicates that much of industry does not collect the necessary data to build measurement programs. For the most part, our earlier survey [Zelkowitz98], the composite scores, and the results in Tables 3.1 to 3.3 are compatible. In the earlier survey, papers studied from 1995 used case study and lessons learned equally, followed by simulation at half that number. In Table 3.3, the student population considering the research techniques ranked case study as practical in six of the seven questions. The industrial group (Table 3.2) selected either measurement or case study as practical for six of the seven questions, but the researchers find case study either impractical or neutral. Case study requires collection of data and measurement. It appears that the industry population values these measurement techniques as important, cost is a significant driver to industry, measurement techniques are perceived as too expensive. Better methods and tools for aiding measurement techniques are required to address industry concerns and to make the techniques more acceptable to researchers.

Given that industry is most concerned with internal validity, better tools are needed to aid the research community so that external validity can be conveyed more effectively to the industrial community. This would limit the effects of the "silver bullet" solution to complex problems. Studies are needed to identify:

1. What are the primary drivers that affect applicability in different environments?
2. How do you measure the effectiveness of a new method in a different environment?

Some of the results obtained here may be viewed as obvious, but we believe that these opinions have not been quantified previously. The industrial and the research community do look at method validation for different purposes, so it is not too surprising that one does not share the beliefs of the other. This leads to conflicts when one group does not provide or use the results of the other.

Given the set of techniques described here, it would aid both communities if those techniques near the top of the rankings had better tool support. Measurement is clearly important to the industrial professional, so less expensive data collection methods are needed. Tools for collecting defect data or analyzing defect and resource data are needed. Tools to better evaluate case studies would help. How to deal with the high cost and poor perception of the replicated experiment needs to be further studied.

In this paper, as with our earlier survey of the research literature, we have tried to understand the process that organizations use to evaluate new technologies and transition them into industrial use. We haven't solved the significant technology transition problems with this survey, but we do believe we have indicated where further research is needed and why some of the current problems in technology transition exist. We need to further understand both cultures in order to determine which technique can best enable industry to make intelligent choices on which new technology to use and, we emphasize the need for research to develop the methods and tools to make these techniques practical..

Acknowledgments

We thank Dr. Nien Zhang for his suggestions regarding statistical methods for viewing this data.

References

[Brown96] Brown A. W. and K. C. Wallnau, A framework for evaluating software technology, *IEEE Software*, (September, 1996) 39-49.

[Fenton94] Fenton N., S. L. Pfleeger, and R. L. Glass, Science and substance: A challenge to software engineers, *IEEE Software,* Vol. 11, No. 4, 1994, 86-95.

[Daly97] Daly, J., K. El Emam, and J. Miller, Multi-method research in software engineering, 1997 IEEE Workshop on Empirical Studies of Software Maintenance (WESS '97) Bari, Italy, October 3, 1997.

[Tichy95] Tichy W. F., P. Lukowicz, L. Prechelt, and E. A. Heinz, Experimental evaluation in computer science: A quantitative study, *J. of Systems and Software* Vol. 28, No. 1, 1995 9-18.

[Tichy98] Tichy, W., Should computer scientists experiment more?, *Computer*, Vol.31, No.5, 1998, pp. 32-40.

[Zelkowitz97] Zelkowitz M. and D. Wallace, Experimental validation in software engineering, *Information and Software Technology*, Vol. 39, 1997, 735-743.

[Zelkowitz98] Zelkowitz M. and D. Wallace, Experimental models for validating technology, *Computer*, Vol.31, No.5, 1998, 23-31.

APPENDIX 1 -- Types of Research Validation

1. **Case study** - a project is monitored and data collected over time. Data collection is derived from a specific goal for the project. A certain attribute is monitored (e.g., reliability, cost) and data is collected to measure that attribute.

2. **Dynamic analysis** - a product is executed for performance. Many methods instrument the given product by adding debugging or testing code in such a way that features of the product can be demonstrated and evaluated when the product is executed.

3. **Legacy data** - data from previous projects is examined for understanding in order to apply that information on a new project under development. Available data includes all artifacts involved in the product, e.g., the source program, specification, design, and testing documentation, as well as data collected in its development.

4. **Lessons-learned** - qualitative data from completed projects is examined. Lessons-learned documents are often produced after a large industrial project is completed. A study of these documents often reveals qualitative aspects which can be used to improve future developments.

5. **Literature search** - previously published studies are examined. It requires the investigator to analyze the results of papers and other documents that are publicly available (e.g., conference and journal articles).

6. **Project monitoring** - collect and store development data during project development. The available data will be whatever the project generates with no attempt to influence or redirect the development process or methods that are being used.

7. **Field study** - A field study may examine data collected from several projects (e.g., subjects) simultaneously. Typically, data are collected from each activity in order to determine the effectiveness of that activity. Often an outside group will monitor the actions of each subject group, whereas in the case study model, the subjects themselves perform the data collection activities.

8. **Replicated experiment** - develop multiple versions of product. In a replicated experiment several projects are staffed to perform a task in multiple ways. Control variables are set (e.g., duration, staff level, methods used) and statistical validity can be more applied. This is the "classical" scientific experiment where similar process is altered repeatedly to see the effects of that change.

9. **Simulation** - execute product with artificial data. Related to dynamic analysis is the concept of simulation. We can evaluate a technology by executing the product using a model of the real environment. We hypothesize, or predict, how the real environment will react to the new technology.

10. **Static analysis** - examine structure of developed product. This is a special case of studying legacy data except that we centralize our concerns on the product that was developed, whereas legacy data also included development process measurement.

11. **Synthetic environment** - replicate one factor in laboratory setting. In software development, projects are usually large and the staffing of multiple projects (e.g., the replicated experiment) in a realistic setting is usually prohibitively expensive. For this reason, most software engineering replications are performed in a smaller artificial setting, which only approximates the environment of the larger projects.

12. **Theoretical analysis** - uses logic to validate a theory; validation consists of logical proofs derived from a specific set of axioms.

APPENDIX 2 -- Types of Industrial Evaluation

1. **Case study** -- Sample projects, typical of other industrial developments for that organization, are developed, where some new technology is applied and the results of using that technology are observed.

2. **Data mining** -- Completed projects are studied in order to find new information about the technologies to develop those projects.

3. **Demonstrator projects** -- Multiple instances of an application, with essential features deleted, are built in order to observe behavior of the new system.

4. **Feature benchmark** -- Alternative technologies are evaluated and comparable data are collected. This is usually a "desk study" using documentation on those features.

5. **Field study** -- An assessment is made by observing the behavior of several other development groups over a relatively short time.

6. **Measurement** -- Data is continually collected on development practices. This data can be investigated when a new technology is proposed.

7. **Pilot study** - A sample project that uses a new technology. This is generally a smaller application (than a case study) before scaling up to full deployment, but is more complete than a demonstration project.

8. **Research literature** -- Information is obtained from professional conferences, journals, and other academic sources of information.

9. **Shadow (Replicated) project** -- One or more projects duplicate another project in order to test different alternative technologies on the same application.

10. **State of the art** -- Using a new technology that is based upon purchaser or client desires or government rules to only use the latest or best technology.

11. **Survey** -- Experts in other areas (e.g., other companies, academia, other projects) are queried for their expert opinion of the probable effects of some new technology.

12. **Theoretical analysis** -- Basing an opinion on the validity of the mathematical model of a new technology.

13. **Vendor opinion** -- Vendors (e.g, through trade shows, trade press, advertising, sales meetings) promote a new technology.

Experimental Models of Technology Transfer

Dolores R. Wallace
Information Technology Laboratory
National Institute of Standards and Technology
&
Marvin V. Zelkowitz
University of Maryland and
Fraunhofer Center - Maryland

Acknowledgements

- This activity also aided by:
  - David Binkley, Loyola College, Baltimore, Maryland

Goal of activity

- To understand how experimental validation of new technology plays a role in technology transfer
  - Models of experimental validation - 1996
  - Perceptions among research and practitioner communities - 1998

Research validation methods

- Case study
- Dynamic analysis
- Field study
- Legacy data
- Lessons learned
- Literature search
- Project monitoring
- Replicated
- Simulation
- Static analysis
- Synthetic
- Theoretical analysis

Industrial transition methods

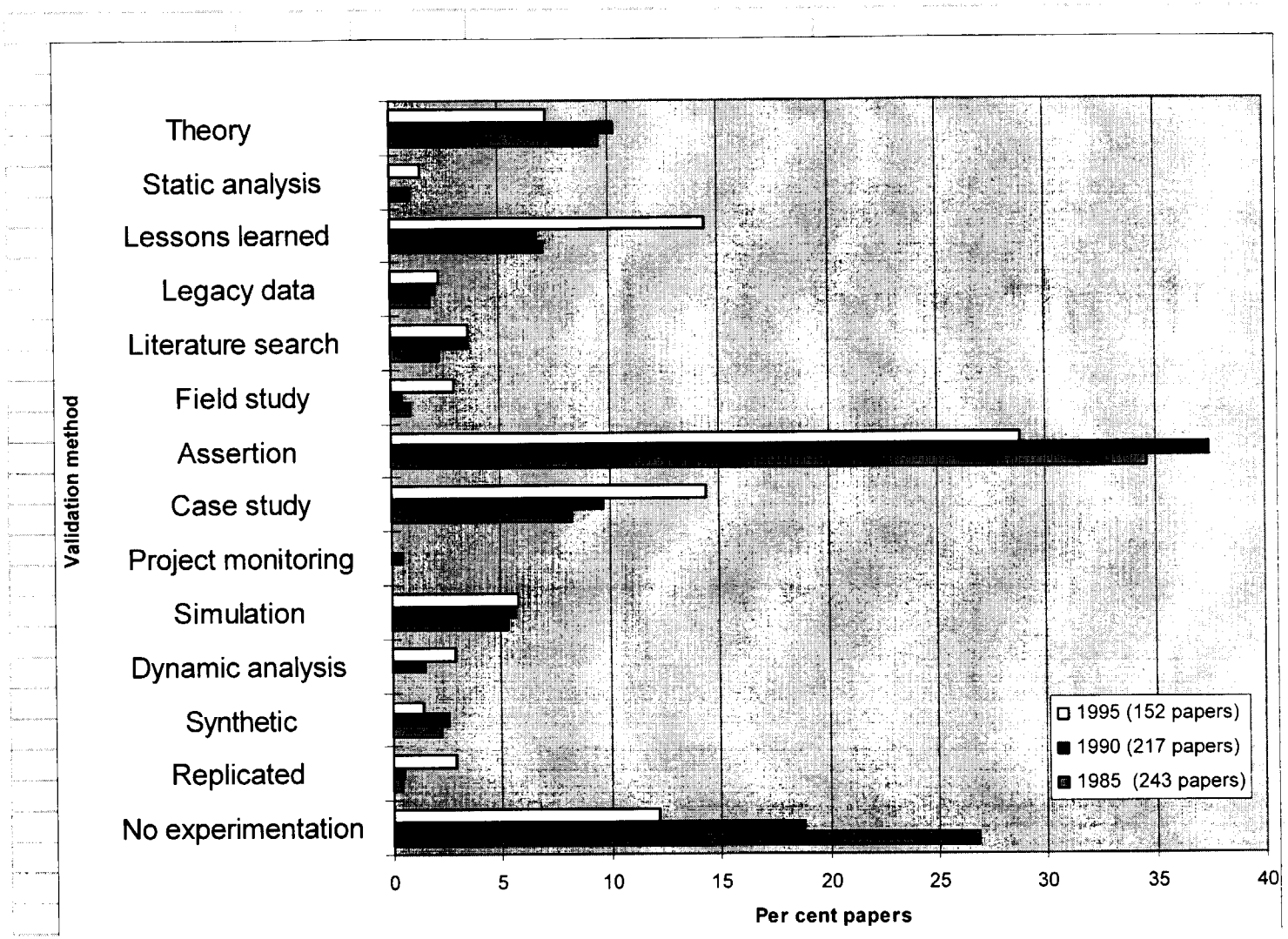
- Case study
- Data mining
- Demonstrator project **
- Feature benchmark
- Field study *
- Measurement
- Pilot study **
- Research literature
- Shadow (replicated) proj.
- State of the art
- Survey *
- Theoretical analysis
- Vendor opinion
- Training +
- People +

Evaluation of 612 journal papers

(May, 1998 IEEE Computer)

Method	ICSE	Soft.	TSE	ICSE	Soft.	TSE	ICSE	Soft.	TSE	
Not applicable	6	6	3	4	16	2	5	7	1	50
No experimentation	13	10	38	7	8	22	7	3	7	115
Replicated	1	0	0	0	0	1	1	0	3	6
Synthetic	3	1	1	0	1	4	0	0	2	12
Dynamic analysis	0	0	0	0	0	3	0	0	4	7
Simulation	2	0	10	0	0	11	1	1	6	31
Project monitoring	0	0	0	0	1	0	0	0	0	1
Case study	5	2	12	7	6	6	4	6	10	58
Assertion	12	13	54	12	19	42	4	14	22	192
Field study	1	0	1	0	0	1	1	1	2	7
Literature search	1	1	3	1	5	1	0	3	2	17
Legacy data	1	1	2	2	0	2	1	1	1	11
Lessons learned	7	5	4	1	4	8	5	7	8	49
Static analysis	1	0	1	0	0	0	0	0	2	4
Theory	3	1	18	1	0	19	3	0	7	52
Yearly totals	56	40	147	35	60	122	32	43	77	612

Relative use of each method



1998 Study

- Determine perceptions of which methods should be most effective
- Perform survey over internet
- Obtain results from researchers and practitioners alike
- Concept based upon WESS '97 paper by J. Daly, K. El Emam, and J. Miller

Sample populations

- Invitations to participate randomly generated
- Research population - U.S.-based authors from research software engineering conferences with email addresses
- Industrial population - U.S.-based authors from industrial software engineering conferences with email addresses
- Professional MS degree students with industrial experience, filled out research form
- 150 invitations sent to each group, about 50 agreed and were sent form, about half returned form
- 44 of 46 students returned form
- Total data: 62 research forms, 25 industrial forms

Survey questions

Rate from 1-20 relative difficulty of each method according to the 7 criteria (1 easy, 20 impossible, 10 maximum practical):

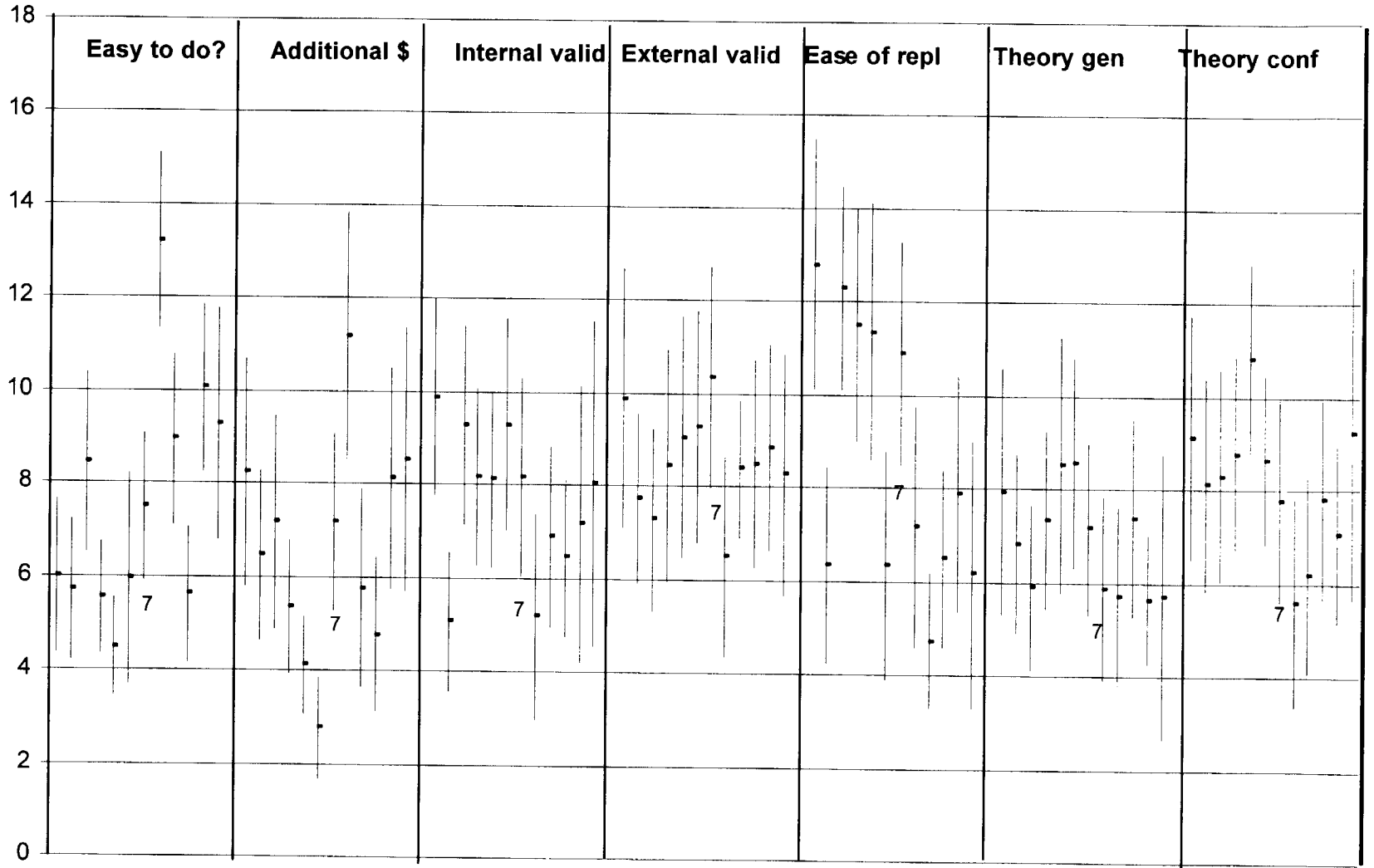
- How easy is it to use this method in practice?
- What is cost of adding one extra subject to study?
- What is the internal validity of the method?
- What is the external validity of the method?
- What is the ease of replication?
- What is the potential for theory generation?
- What is the potential for theory confirmation?

Summary of participants

Sample	Survey	Sample size	Years exper.	Acad. Pos.	Indust. R&D	Indust. Devel.	Other (e.g., Consult.)
1 (Research)	Research	18	18.6	9	3	3	3
2 (Industry)	Industry	25	19.1	0	5	8	12
3 (Students)	Research	44	6.6	1	5	27	11

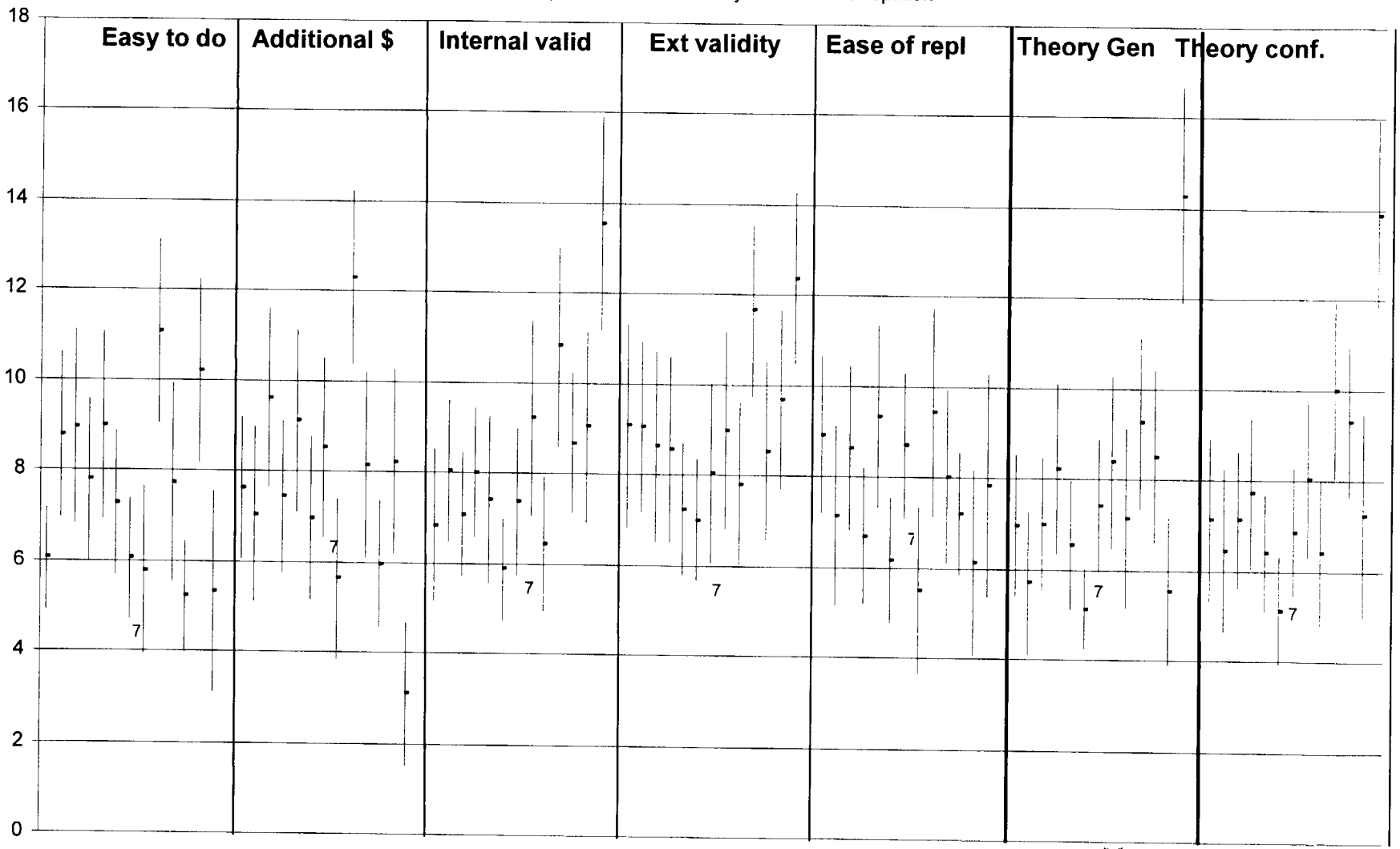
Research group results

1-case study 2-dynamic analysis 3-field study 4-lessons learned 5-legacy data 6-project monitoring 7-literature search 8-replicated experiment 9-simulation 10-static analysis 11-synthetic study 12-theoretical analysis



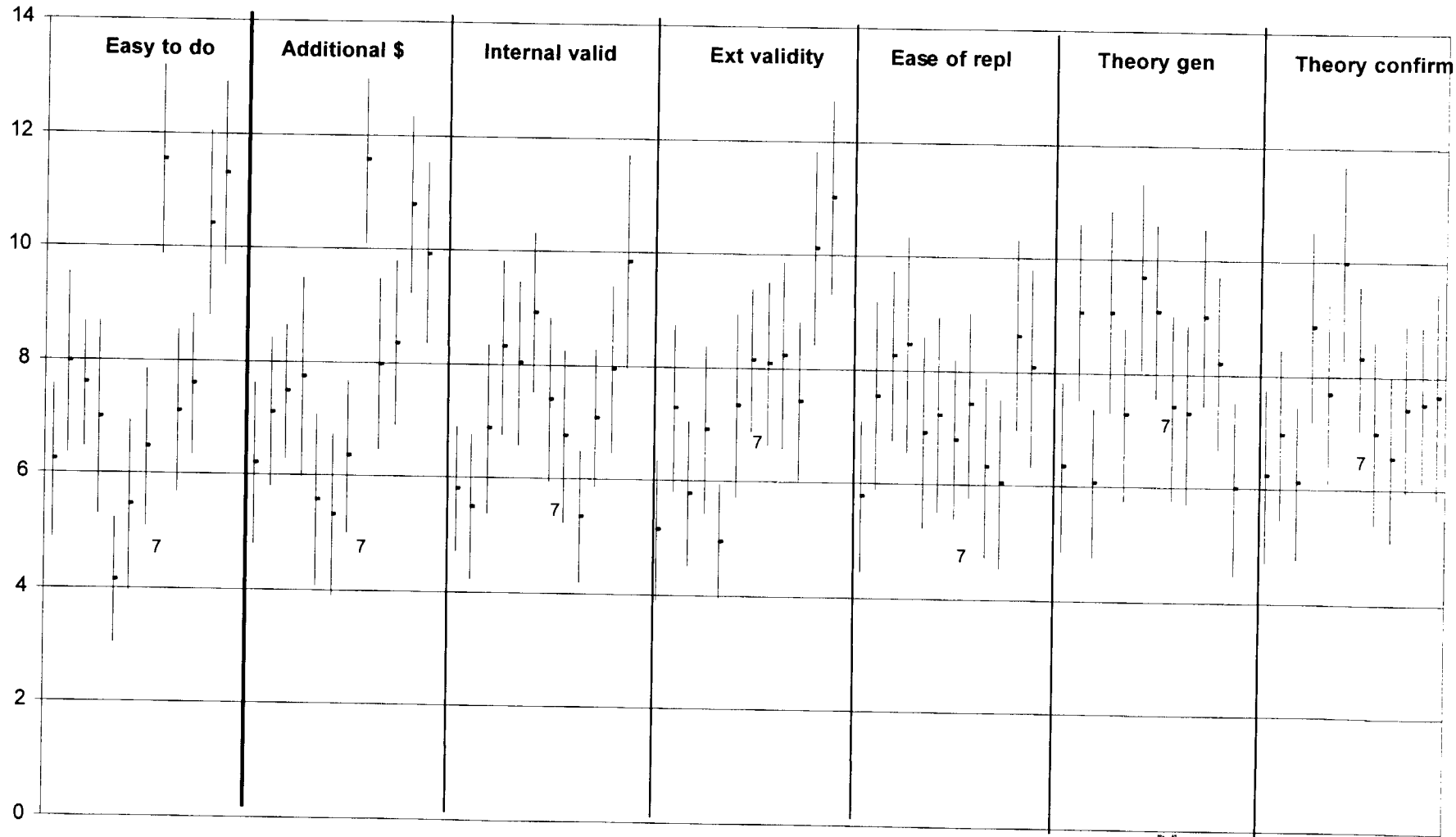
Industrial group results

1-case study 2-data mining 3-demonstrator projects 4-feature benchmark 5-field study 6-measurement 7-pilot study 8-research literature 9-shadow(replicated) project 10-state of the art 11-survey 12-theoretical analysis 13-vendor opinion



Student group results

1-case study 2-dynamic analysis 3-field study 4-lessons learned 5-legacy data 6-project monitoring 7-literature search 8-replicated experiment 9-simulation 10-static analysis 11-synthetic study 12-theoretical analysis



Technique Distribution

- Practical techniques - Better than average - Each method whose upper confidence interval was below the average value for all techniques
- Impractical - Worse than average - Each method whose lower confidence interval was above the average value for all methods
- Neutral - All other methods

Research Technique Distribution -Researchers

	Easy	Addit. $	Int. Val.	Ext. Val. Theory Gen.	Ease of Repl.	Theory Conf.
Practical	Dyn. anal Les. learned Legacy data Static anal.	Legacy data Proj. mon. Static anal.	Dyn. anal. Replicated		Dyn. anal. Simulation Static anal.	Replicated
Impractical	Replicated Synthetic	Replicated	Case study		Case study Field study Les. learned	Legacy data

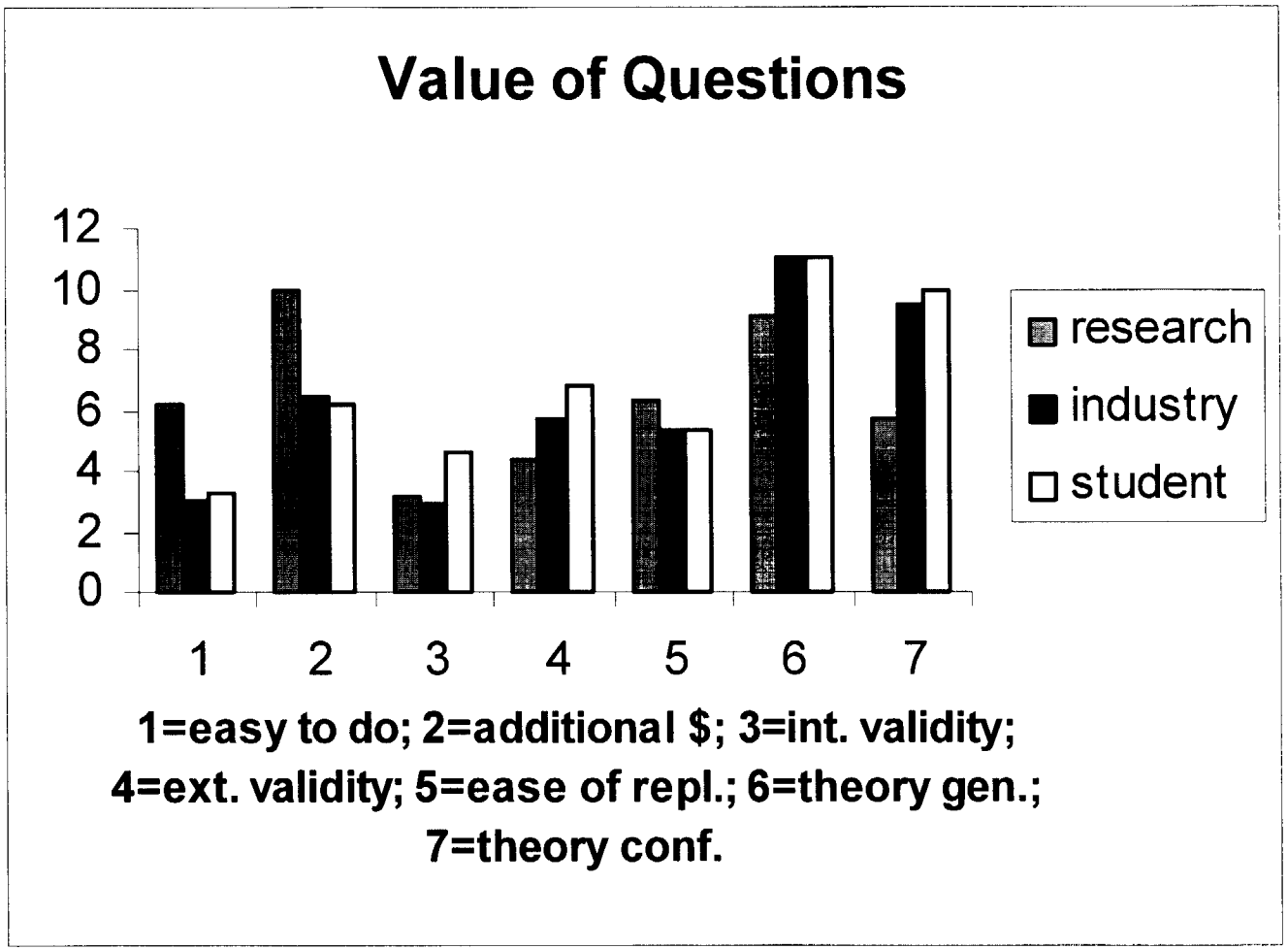
Research Technique Distribution - Industry(Students)

	Easy	**Addit. $**	**Int. Val.**	**Ext. Val.**	**Ease of Repl.**	**Theory Gen.**	**Theory Conf.**
Practical	Case study Legacy data Proj. mon.	Case study Legacy data Proj. mon. Lit. search	Case study Dyn. anal. Simulation	Case study Legacy data	Case study	Case study Field study Theory anal.	Field study
Impractical	Replicated Synthetic Theory anal.	Replicated Synthetic Theory anal.	Proj. mon. Theory anal.	Synthetic Theory anal.		Proj. mon.	Proj. mon.

Industry Technique Distribution - Industry

	Easy	Addit $.	Int. Val.	Ext. Val.	Ease of Repl.	Theory Gen.	Theory Conf.
Practical	Case study Pilot study Survey Vendor opinion	Lit. Res. Survey Vendor opinion	Measure	Field study Measure	Measure Lit. res.	Data mining Measure Theory anal.	Field study Measure
Impractical	Replicated	Replicated	State of art Vendor opinion	State of art Vendor opinion		Vendor opinion	State of art Vendor opinion

Relative importance of criteria
(when making a decision)



Composite measure
(sum individual criteria)

Sample 1 Research group		Sample 3 Student group		Sample 2 Industry group	
Simulation	288	Case study	284	Measurement	258
Static analysis	292	Legacy data	314	Data mining	305
Dynamic analysis	298	Field study	315	Theoretical analysis	324
Project monitoring	301	Simulation	333	Literature research	325
Lessons learned	339	Dynamic analysis	355	Case study	326
Legacy data	345	Static analysis	361	Field study	327
Synthetic study	346	Literature search	370	Pilot study	329
Theoretical analysis	348	Replicated experiment	387	Feature benchmark	338
Field study	363	Project monitoring	388	Survey	343
Literature search	367	Lessons learned	391	Demonstrator project	345
Replicated experiment	368	Theoretical analysis	405	Replicated project	361
Case study	398	Synthetic study	418	State of art	407
				Vendor opinion	469

Composite measures

- For the research community, tools-based techniques dominate the rankings
- For the student community, those techniques that confirm a technique in the field dominate the rankings
- For the industrial group, measurement and data mining clearly dominates the ranking even though rarely done in practice
- For the industrial group, state of the art and vendor opinion came last, so why resort to that so often?

Current plan

- Develop new survey instrument on actual experiences in transferring a technology
- Working with Shari Lawrence Pfleeger
- Obtain large industrial sample
  - Models of experimental validation - 1996
  - Perceptions among research and practitioner communities - 1998
  - Experiences of experimental validation - planned

Conclusions

- Conflicting views: Researchers use tools; industry wants field experiences- Need for better understanding of needs of both communities (publications vs. practicality)
- Measurement viewed as important, but anecdotal evidence that not practiced much
- It would aid both communities if those techniques near the top of the rankings had better tool support
- Need evaluation process to determine most effective method for a given new technology

An Adaptation of Experimental Design to the Empirical Validation of Software Engineering Theories

55-61

N. Juristo, A.M. Moreno

Facultad de Informática - Universidad Politécnica de Madrid -
Campus de Montegancedo s/n, 28660 Madrid
Tel.: + 34 91 336 69 22; Fax: + 34 91 336 69 17
{natalia, ammoreno}@fi.upm.es

Abstract

This paper has two objectives. Firstly, it seeks to promote discussion and debate about the need to encourage experimentation of the claims in the field of software engineering. The software community's lack of concern for the need for the aforesaid experimentation is slowing down adoption of new technology by organizations unfurnished with objective data that show the benefits of the new artifacts to be introduced. This situation is also leading the introduction of new software technology to be considered as a risk, because, as it has not been formally validated beforehand, its application can cause disasters in user organizations. The second objective is to present a formal method of experimentation in SE, based on the experimental design and analysis techniques used in other branches of science.

1. Introduction

Companies are continuously developing new, increasingly complex and, ultimately, more expensive software systems. This should be a condition for applying the range of development artifacts in a reliable manner. Paradoxically, however, real-world developments are often used as a culture medium for validating these artifacts, with the ensuing risks. There is no denying, unfortunately, that the models and theories outputted by Software Engineering (SE) research are not checked against reality as often as would be necessary to assure their validity for use in software construction. This can lead to justified distrust when applying the new solutions developed at laboratories or research centers in industry.

It is, therefore, essential to apply a process of experimental testing to validate any contribution made to SE. This paper seeks to highlight the need for an empirical validation of all artifacts used in SE, and then proposes an approach to introduce this based on experimental design techniques, widely used in other fields of science and engineering. Other researchers, including Basili [Basili, 86] and Pfleeger [Pfleeger, 95], have published work on experimental design and SE. In this paper, we aim to address in detail particular points, such as the parameters to be controlled in a SE experiment, and will set out several examples of how different types of experimental design can be applied to SE.

So as show the lack of empirical validation in the field of SE, the authors have compared what we have called *the essence of the scientific method* with SE research. The essence of the scientific method relates to certain characteristics common to the different methods of research with regard to the manner of attaining new knowledge. These common features can be divided into the following activities:

- **Interaction with reality**, which involves obtaining facts from reality. It can be performed by means of observation, where researchers merely perceive facts from the outside, or by means of experimentation, where researchers subject the object to new conditions and observe the reactions.
- **Speculation**, where researchers think about the perception obtained from the outside world. The results of this thinking range from a mere description of particular cases, through hypotheses and models, to general laws and theories.
- **Checking ideas against reality** in order to assure the truth of the speculations. It can safely be said that it is this stage that lends research its scientific value, as the stages of interacting with reality and speculation occur in other intellectual disciplines far from being considered scientific; for example, philosophy, religion, politics, etc. A branch of human knowledge attains the status of scientific when speculations are verifiable and, therefore, valid (although this status is always held provisionally until contradicted by a new reality). Remember that engineering fields depend on scientific knowledge to

build their artifacts.

When comparing the essence of the scientific method and research in SE, there are a series of discrepancies, including importantly the lack of emphasis on the experimental validation activity. In fact, present scientific progress in the software community appears to be based on natural selection. That is, researchers throw their lucubrations into the arena almost untested. After a few years or decades, theoretically, the fittest survives. Note the risk involved in this manner of scientific progress, as fashion, researcher credibility, etc., also play a prominent role in science. This way of selecting valid knowledge involves important risks when industry applies this new knowledge.

Statements claiming that SE experimentation is not needed can be heard frequently in SE. One of the arguments is that the "Romans built bridges and were not acquainted with the scientific method". Obviously, humans can generate valid knowledge by means of trial and error. However, this approach is longer and more chancy than the scientific method. If a critical software system fails and causes a disaster, could we say that we in SE prefer the old trial-and-error approach rather than experimental validation as called for by the scientific method? Another justification used to refute SE experimentation is based on trusting in intuition. Several examples can be used to reject this statement, for example, the fact that small software components are proportionally less reliable than larger ones, as reported by Basili [Basili, 94] among others. In [Tichy, 98] the author presents some arguments traditionally used to reject the usefulness of experimentation in this area with the corresponding refutation.

Although there are some experimental studies in the computer science literature [Prechelt, 98] [Frankl, 93] [Seaman, 98] [Iyer, 90], this is not the general rule. The want of experimental rigor in SE has already been stressed by authors like Zelkowitz [Zelkowitz, 98] or Tichy [Tichy, 93] [Tichy, 95], who base this affirmation on a study of the papers published in several system-oriented journals. Surveys such as Zelkowitz's and Tichy's tend to validate the conclusion that the SE community can do a better job in reporting its results, making them more trustworthy and thus making it easier for industry to adopt the new research results.

2. Experimental Design for Software Engineering

Once that the need for empirical validation in SE has been assumed, the authors propose an approach to introduce it based on experimental design techniques [Box, 78] [Selwyn, 96] [Clarke, 97] [Edwards, 98] used in others fields of science.

Empirical validation can be carried out in several situations : *laboratory* validation of theories, validation at the level of *real projects* and validation by means of *historical data*. Unlike the other two methods, laboratory validation allows greater control of the different parameters that affect software development. Real projects allow data considered to be relevant for the study in question to be collected. Validation using historical data allows researchers to work with data on finished projects, employing the most relevant for the experiment to be conducted. Zelkowitz [Zelkowitz, 98] and Kitchenham [Kitchenham, 96] suggested similar classifications. Zelkowitz groups experimental approaches into three broad categories: *controlled methods, observational methods* and *historical methods*, while Kitchenham refers to these categories of experimentation as *formal experiments, case studies*, and *surveys*. An example of experimentation with real projects is the experience factory proposed by Basili [Basili, 95], historical data have been applied by McGarry [McGarry, 97] among others, and formal experiments have been studied by Pfleeger [Pfleeger, 95] in the DESMET project.

In this paper, we focus on formal experiments and present an in-depth study of the application of experimental design to SE empirical validation, placing special emphasis on the adaptation of experimental design terminology to SE. Table 1 summarizes the above-mentioned experimentation process. Table 2 describes the application of experimental design concepts to SE. Table 3 shows the value of some of the experimental design concepts for SE experimentation. Finally, Table 4 presents a summary of the experimental design techniques that can be applied.

Phase of the experiment	Description
Defining the Objectives of the Experiment.	The mathematical techniques of experimental design demand that experiments produce quantitative results. Therefore formal experimentation in SE requires quantifiable hypotheses. This hypothesis will be usually expressed in terms of a metric of the software product developed using the software artifact to be analyzed or of the development process where this artifact has been applied.
Designing the Experiment	In order to plan experimentation in SE according to experimental design guidelines, its terminology has to be applied to SE. See table 2 with the terminology employed in experimental design for generic experimentation, and its application to experiments in SE. The next step is to select the experimental design technique. This technique will determine how many experiments are required, how many times each experiment has to be repeated and what data we need to output to ascertain the validity of the conclusions. There are different techniques of experimental design depending on the aim of the experiment, the number of factors, the levels of the factors, etc. Table 4 shows a brief summary of the most commonly used experimental design techniques.
Executing Experiments	The software engineer is now ready to execute the experiments indicated as a result of the preceding design stage, measuring the response variables at the end of each experiment.
Analyzing Results	This stage is also called Experimental Analysis. The software engineer will quantify the impact of each factor and each interaction between factors on the variation of the response variable. This is what is referred to (according to experimental design terminology) as "the statistical significance of the differences in the response variable due to the different levels of each factor". • If there is no statistical significance, the variation in the response variable can be put down to chance or to another variable not considered in the experiment. • If there is statistical significance, the variation in the response variable is due to the fact that a certain level (or combination of levels of different factors) causes improvements in the response variable. When we have understood the impact, we can ascertain which alternative of which factor significantly improves the value of the response variable. Depending on the experimental design technique applied in the preceding stage, a different statistical technique must be used to achieve the above objective. This is not the place to expound the underlying mathematics of experimental analysis. Interested readers are referred to the references already mentioned. Section 3 shows some examples of SE experiments illustrating different experimental design and analysis techniques.

Table 1. Phases of the Experimental Design Process used for SE Experiments

Concept	Description	Application in SE
Experimental unit	Entity used to conduct the experiment	Software projects
Parameters	Characteristic (qualitative or quantitative) of the experimental unit	See table 3
Response variable	Datum to be measured during the experimental unit	See table 3. Note there are no response variables relating to the "problem". This is because response variables are data that can be measured *a posteriori*, that is, once the experiment is complete. In the case of SE, the experiment involves development (in full or in part) of a software system to which particular technologies are applied. The characteristics of the *problem* to be solved are the experiment input data, that is, they stipulate how it will be performed. As such, they are parameters and factors of the experiment. However, they are not experimental output data that can be measured and, thus, do not generate response variables.
Factor	Parameter that affects the response variable and whose impact is of interest for the study	Factors are chosen from the parameters in table 3. Factors have different values during the experiment
Level	Possible values or alternatives of the factors	Values of factors in table 3
Interaction	The effect of one factor depends on the level of another	Relations between the parameters in table 3; for example, problem complexity and product complexity
Replication	Repetition of each experiment to be sure of the measurement taken of the response variable	Repeatability in SE must be based on analogy, not on identity; the different experiments will consist of similar problems, similar processes, similar teams, etc.
Design	Specification of the number of experiments, selection of factors, combinations of levels of each factor for each experiment and the number of replications per experiment	The design will indicate the number of software projects, factors and their alternatives that will be used during experimentation, as well as the number of replications of the experiments, based on analogy.

Table 2. Application of experimental design concepts to SE

PARAMETERS			
PROBLEM (User need)	PROCESSES of construction employed	PERSONS (team of developers)	PRODUCT
– Definition (poorly/well defined problem) – Need volatility (very/hardly/non volatile need) – Ease of understanding (problem well/poorly/fairly well understood by developers) – Problem complexity – Problem type (data processing, knowledge use, etc.), – Problem-solving type (procedural, heuristic, real-time problem solving, etc.) – Domain (aeronautics, insurance, etc.) – User type (expert, novice, etc.)	– Maturity – Description (set of phases, activities, products, etc.) – Relationship between members (definition of interrelations between team members) – Automation (in which phases or activities tools are used) – Risks	– Number of members – Division by positions (no. of software engineers, programmers, project managers, etc.) – Years of experience of each member in development – Experience of each member in the problem type – Experience of each member in the software process applied – Background of each member (discipline of origin) – Type of relationship between members (all in the same building, same town, subcontracts, etc.)	– Type of life cycle to be followed – Software type (OO, databases, real time, expert system, etc.) – Size – Complexity – Architecture/Organization – Hardware platform – Interaction with other software – Processing conditions (batch, on-line, etc.) – Security requirements – Response-time requirements – Documentation required – Help required
RESPONSE VARIABLES			
PROBLEM	PROCESS	PERSONS	PRODUCT
	– Schedule deviation – Budget deviation – Compliance with construction process – Products obtained (do they comply with the process stipulations?)	– Productivity – User satisfaction – usability – usefulness	– Correctness of products obtained (no. of errors, etc.) – Validity of the products (compliance with customer expectations) – Portability, Maintainability, Extendibility, Performance, Flexibility, Interoperability,…

Table 3. Proposal of Parameters and Response Variables for SE research

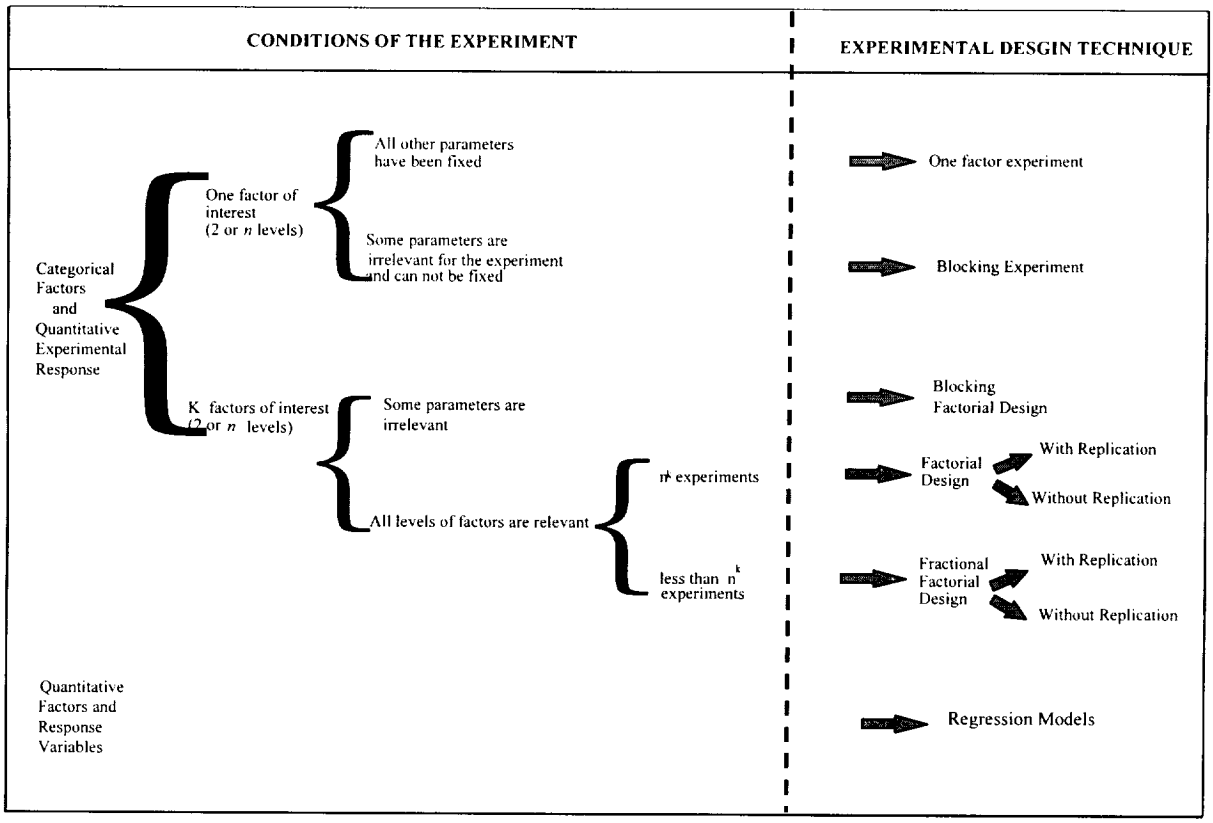


Table 4. Different Experimental Design Techniques

3. Example of SE Experiments using Experimental Design

This section presents two examples of possible SE experiments employing the experimental design process described in Table 1. Depending on the experimental desgin techinque used, different analysis methods must be applied. During the experimental analysis phase, we will not enter into a detailed justification of all the mathematical calculations; our objective is simply to give readers a taste of what sort of work could be performed during an experimentation in SE, avoiding the tiresome, though simple, calculations called for by experimental analysis.

3.1. One Factor Experiment

Suppose we are researching on a CASE tool, and we think it will increase programmers productivity. We will compare this tool with two other tools widely used in industry and each experiment will be repeated five times, in order to consider experimental errors. The response variable will be *programmers productivity* (lines of code/person-day) and all other parameters of table 3 will be fixed. This is an example of one factor experiment. This kind of experimental design is used to determine the best choice of k alternatives (in our case of three alternatives).

Table 5 shows the fifteen observations of the response variable (column Z contains the values for the new tool).

R	V	Z
144	101	130
120	144	180
176	211	141
288	288	374
144	72	302

Table 5. Value of the response variables

The analysis if this experiment is shown in table 6. From this table we can know that the mean value of productuvity of a CASE tool is 187,7 lines/person-day. The effects of tools R, V and Z are -13,3, -24,5 and 37,7, respectively. That means that tool R provides 13,3 lines less than the mean, tool V provides 24,5 lines less than the mean, and tool V provides 37,7 lines more than the mean.

	R	V	Z	
	144	101	130	
	120	144	180	
	176	211	141	
	288	288	374	
	144	72	302	
Sum of the column	$\sum Y_{\bullet 1} = 872$	$\sum Y_{\bullet 2} = 816$	$\sum Y_{\bullet 3} = 1127$	$\sum Y_{\bullet\bullet} = 2815$
Mean of the column	$\bar{Y}_{\bullet 1} = 174.4$	$\bar{Y}_{\bullet 2} = 163.2$	$\bar{Y}_{\bullet 3} = 225.4$	$\mu\bar{Y}_{\bullet\bullet} = 187.7$
Effect of the column	$\alpha_1 = \bar{Y}_{\bullet 1} - \bar{Y}_{\bullet\bullet} = -13.3$	$\alpha_2 \bar{Y}_{\bullet 2} - \bar{Y}_{\bullet\bullet} = -24.4$	$\alpha_3 \bar{Y}_{\bullet 3} - \bar{Y}_{\bullet\bullet} = 37.7$	

Table 6. Data from the experimental analysis of the example

The second step involves calculating the sum of the squared errors (SSE) in order to estimate the variance of the errors and the confidence interval for effects. For that aim each observation will be divided in three parts: the grand mean, the effect of the tool, and the residuals. For each part we have used a matrix notation.

$$\begin{bmatrix} 144 & 101 & 130 \\ 120 & 144 & 180 \\ 176 & 211 & 141 \\ 288 & 288 & 374 \\ 144 & 72 & 302 \end{bmatrix} = \begin{bmatrix} 187.7 & 187.7 & 187.7 \\ 187.7 & & \\ 187.7 & & \\ 187.7 & & \\ 187.7 & & 187.7 \end{bmatrix} + \begin{bmatrix} -13.3 & -24.5 & 37.7 \\ & & \\ & & \\ & & \\ -13.3 & -24.5 & 37.7 \end{bmatrix} + \begin{bmatrix} -30.4 & -62.2 & -95.4 \\ -54.4 & -19.2 & -45.4 \\ 1.6 & 47.8 & -84.4 \\ 113.6 & 124.8 & 148.6 \\ -30.4 & -91.2 & 76.6 \end{bmatrix}$$

$$SSE = \sum_{i=1}^{r}\sum_{j=1}^{a} e_{ij}^2 = (-30,4)^2 + (-54,4)^2 + \ldots + (76,6)^2 = 94.365,20$$

Next step is calculating the variation in the response variable due to the factor and to the experimental error. For that aim we calculate the sum of squares total (SST).

$$SST = r\sum_{j} \tau_j^2 + SSE = 5\,((-13,3)^2 + (-24,5)^2 + (37,6)^2) + 94.365,2 = 105.357,3$$

The percentage of variation in the response variable explained by CASE tools is 10,4% (10.992,13/105.357,3). The rest of the variation 89,6% is due to experimental errors. That means that the experiment has not been planned properly.

In order to determine whether the variation of 10,4% in the productivity has statistical significance we have to use the ANOVA (Analysis Of VAriance) technique, with the F-test function and table (this table is not included in the paper, readers can find them in the bibliography of experimental design mentioned above). The technique seeks to compare the contribution of the factor to the variation in the response variable with the contribution of the errors. If the variation due to errors is high, a factor that explains a high variation in the response variable might has not statistical significance. In order to determine the statistical significance we will compare the computed F-value with the value got from the F-table, as shown in table 7.

Table 8 shows the ANOVA analysis for our example. The calculated F-value is smaller than the one got from the F-table. Therefore, we can, again, conclude that the difference in productivity is mainly due to experimental errors instead of to the CASE tools. In that sense, we can state that neither tool provides more productivity than the others.

COMPONENT	SUM OF SQUARES	PERCENTAGE OF VARIATION	DEGREES OF FREEDOM	MEAN SQUARE	F-COMPUTED	F-TABLE
Y	$SSY = \sum Y_{ij}^2$		ar			
$\bar{Y}_{..}$	$SSO = ar\mu^2$		1			
$Y - \bar{Y}_{..}$	$SST = SSY - SSO$	100	ar-1			
A	$SSA = r\sum \alpha_i^2$	$100\left(\frac{SSA}{SST}\right)$	a-1	$MSA = \frac{SSA}{a-1}$	$\frac{MSA}{MSE}$	$F_{[1-\alpha;a-1;a(r-1)]}$
e	$SSE = SST - SSA$	$100\left(\frac{SSE}{SST}\right)$	a(r-1)	$MSE = \frac{SSE}{a(r-1)}$		

$$S_e = \sqrt{MSE}$$

Table 7. ANOVA table for one factor experiments

Y	633,639.00					
Y..	528,281.69					
Y-Y..	105.357,31	100.00	14			
A	10.992,13	10.4	2	5496.1	0.7	2.8
Errors	94.365,20	89.6	12	7863.8		

$$S_e = \sqrt{MSE} = \sqrt{7863.77} = 88.68$$

Table 8. ANOVA table for our experiment

3.2. Factorial Design with Replication

Suppose that we have invented a new development paradigm that is completely different from the structured and OO paradigms and want to confirm that our innovation improves development projects. We will centre on *correctness* as the response variable, measured, for example, by the number of faults emerging three months after software deployment. There are a lot of characteristics that have an impact on this response variable: problem complexity, problem type, process maturity, team experience, software complexity, integration with other software, etc. However, all of these will be fixed at an intermediate value (that is, they will be selected as parameters of the experiment), except *development paradigm,* and *software complexity* which will be factors. Each factor will necessarily admit two alternatives to simplify the calculations. According to experimental design guidelines, the factors, labelled with letters, and their alternatives, labelled with level 1 and -1, are listed, as shown in table 9.

FACTOR	NAME	LEVEL -1	LEVEL 1
Paradigm	A	New	OO
Software complexity	B	Complex	Simple

Table 9. Factors and levels of the experiment

We will use a factorial design with replication as all levels of our factors are relevant for the experiment, and we want to consider the experimental errors. In order to evaluate the experimental errors we will repeat each experiment three times, so we will get twelve measurements of the response variable.

Taking the measurements of the response variable and the values assigned to the factors in table 9, the first step of the experimental analysis is to build what is called the sign table. As shown in table 10, the first column of the matrix is labelled I, and it contains all 1s. The next two columns, labelled with the factor names, contain all the possible combinations of -1 and 1. The fourth column is the product of the entries in columns A and B. The twelve observations are then listed in column Y. The entries in column I are then multiplied by those in last column, and the sum is then entered under column I. The entries in column A are then multiplied by those in last column and the sum is entered under column A. This column multiplication operation is repeated for the remaining columns in the matrix. The sum under each

column is divided by 4 to give the corresponding coefficients of the regression model.

I	A	B	AC	Y	Mean $\bar{Y}$
1	-1	-1	1	(15, 18, 12)	15
1	1	-1	-1	(45, 48, 51)	48
1	-1	1	-1	(25, 28, 19)	24
1	1	1	1	(75,75,81)	77
164	86	38	20		Total
41	21.5	9.5	5		Total/4

Table 10. Sign table for a 2^2 experimentation with replication

The second step involves calculating SSE. Table 11 shows the estimated response and the errors for each of the twelve observations. The estimated value for the response variable is calculated adding the products of the effects (C_0, C_A, C_B, C_{AB}) and the entries (X_A, X_B, X_{AB}) in the sign table.

Effects					Estimated Response	Mean Response			Errors		
i	I 41	A 21.5	B 9.5	AB 5		Y_{i1}	Y_{i2}	Y_{i3}	e_{i1}	e_{i2}	e_{i3}
1	1	-1	-1	1	15	15	18	12	0	3	-3
2	1	1	-1	-1	48	45	48	51	-3	0	3
3	1	-1	1	-1	24	25	28	19	1	4	-5
4	1	1	1	1	77	75	75	81	-2	-2	4

Table 11. Errors in each experiment

The sum the squared errors is:

$$SSE = \sum_{i,j} e_{i,j}^2 = 0^2+3^2+(-3)^2+(-3)^2+ 0^2+3^2+1^2+(-5)^2+(-2)^2+(-2)^2+4^2 = 102$$

Now we want to calculate the variation in the response variable due to each factor or combination of factors, and to the experimental error. For that aim we calculate SST.

$$SST = 2^2 r C_A^2 + 2^2 r C_B^2 + 2^2 r C_{AB}^2 + \sum_{i,j} e_{i,j}^2 = 5{,}547 + 1{,}083 + 300 + 102 = 7{,}032$$

Factor A explains 78,88% (5,547/7,032) of the variation, factor B explains 15,04% and the interaction AB explains 4,27%. The rest of the variation, 1,45%, is a variation non explicated, and therefore, due to experimental errors.

4. Conclusions

In this paper, we presented a possible adaptation of the experimental design techniques used in other branches of science and engineering to perform experiments in SE.

The objective of the paper is not only to present a means of carrying out formal experimentation in SE but also to promote discussion and debate on the need to encourage experimentation of the claims in this field. The software community's lack of concern for the need for the aforesaid experimentation is slowing down adoption of new technology by organizations unfurnished with objective data that show the benefits of the new artifacts to be introduced. This situation is also leading the introduction of new software technology to be considered as a risk, because, as it has not been formally validated beforehand, its application can cause disasters in user organizations.

We are aware that software development's marked economic and commercial nature can be a decisive factor standing in the way of the necessary experimentation, as experimentation does not produce tangible, short-term benefits. The benefit of experimentation will come to fruition in future development projects, and this benefit is difficult to quantify at the time of deciding on experimental feasibility or the number of experiments to be performed. However, as we have already said, experimentation can also stop industry taking unnecessary risks by adopting proposals that have not been satisfactorily tested.

5. References

[Basili, 84] V.R. Basili, B.T. Perricone. *Software Errors and Complexity: An Empirical Investigation.* **Communications of the ACM**, January 1984, pp. 42-52.

[Basili, 86] V.R. Basili, R.W. Selby, D.H. Hutchens. *Experimentation in Software Engineering.* **IEEE Transactions on Software Engineering**, vol. 12 (7), July 1986, pp. 733-743.

[Basili, 95] V. R. Basili. *The Experience Factory and Its Relationship to Other Quality Approaches*, Academic Press Inc., **Adnvances in Computers**, Volume 41, 1995.

[Box, 78] Box, G.E.P., Hunter W.G. and Hunter, J.S. **Statistics for Experiments**. Wiley, New York, (USA), 1978.

[Clarke, 97] Clarke, G.M. and Kempson, R.E. **Introduction to the Design & Analysis of Experiments**. Wiley & Sons, New York (USA), 1997.

[Edwards, 98] Edwards, A.L. **Experimental Design**. Addison-Wesley Educational Publishers, Delaware (USA), 1998.

[Frankl, 93] P.G. Frankl, S.N. Weiss. *An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing.* **IEEE Transactions on Software Engineering**, vol. 19 (8), August 1993.

[Iyer, 90] Iyer, R.K. *Special Section on Experimental Computer Science.* **IEEE Transactions on Software Engineering**, vol. 16 (2), February 1990.

[Kitchenham, 96] Kitchenham, B. *Evaluating Software Engineering Methods and Tools.* Parts 1 to 8. **SIGSOFT** Notes 1996 and 1997.

[McGarry, 97] F. McGarry, S. Burke, W. Deker and J. Haskell. *Measuring Impacts of Software Process Maturity in a Production Environment.* **22nd NASA Workshop on Software Engineering**, Maryland, USA, December 1997, pp. 193-220.

[Pfleeger, 95] Pfleeger, S.L. *Experimental Design and Analysis in Software Engineering.* **Annals of Software Engineering**, vol. 1, 1995, 219-253.

[Prechelt, 98] Prechelt, L and Tichy, W.F. *A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking.* **IEEE Transactions on Software Engineering**, vol. 24 (4), April 1998, 302-318.

[Seaman, 98] Seaman, C.B. and V.R. Basili. *Communication and Organization: An Empirical Study of Discussion in Inspection Meetings.* **IEEE Transactions on Software Engineering**, vol. 24 (7), July 1998, 559-572.

[Selwyn, 96] Selwyn, M.R. **Principles of Experimental Design for the Life Sciences**. CRC Press Inc. (UK) 1996.

[Tichy, 93] Tichy, W.F. *On Experimental Computer Science. International Workshop on Experimental Software Engineering Issues. Critical Assessment and Future Directions.* Proceedings, 1993, 30-32.

[Tichy, 95] Tichy, W.F. et al. *Experimental Evaluation in Computer Science: A Quantitative Study.* **Journal of Systems and Software**, vol. 28, 1995, 9-18.

[Tichy, 98] Tichy, W.F. *Should Computer Scientists Experiment More ?* **IEEE Computer**, May 1998,32-40.

[Zelkowitz, 98] Zelkowitz, M, Wallace, R. *Experimental Models for Validating Technology.* **IEEE Computer**, May 1998, 23-31.

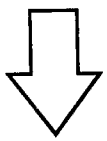
AN ADAPTATION OF EXPERIMENTAL DESIGN TO THE EMPIRICAL VALIDATION OF SOFTWARE ENGINEERING THEORIES

N. Juristo A. M. Moreno
Facultad de Informática
Universidad Politécnica de Madrid
SPAIN

CONTENTS

PROBLEM

Are we sure about the ideas* we use in software development?



Let's think about the VALIDATION OF SE IDEAS

*idea = concept, paradigm, method, technique, tool, etc.

VALIDATION OF SE IDEAS
When? How?

When are we sure that a proposed idea works?

After years of use

How do we validate ideas?

Natural selection

VALIDATION OF SE IDEAS
Implications

— Ideas have not been validated before being proposed to the community.

— Industry uses non-reliable ideas when constructing software.

VALIDATION OF SE IDEAS
Other way?

Is there a different way of doing things?

Yes, through EXPERIMENTATION

Do other scientific and engineering fields validate ideas the same way we do?

No, thet do it through EXPERIMENTATION

WHAT IS EXPERIMENTATION?

— Experimentation makes the difference between science & engineering and other academic disciplines

— Quantitative study of phenomena

— Test/ideas against reality (Empirical Validation)

KINDS OF EMPIRICAL VALIDATION

— Formal Laboratory Experiments

— Case Studies: Real projects

— Use of Historical Data

OUR PROPOSAL

A way to perform Formal Lab Experiments

An Approach to Empirical Validation of SE Ideas in the Laboratory

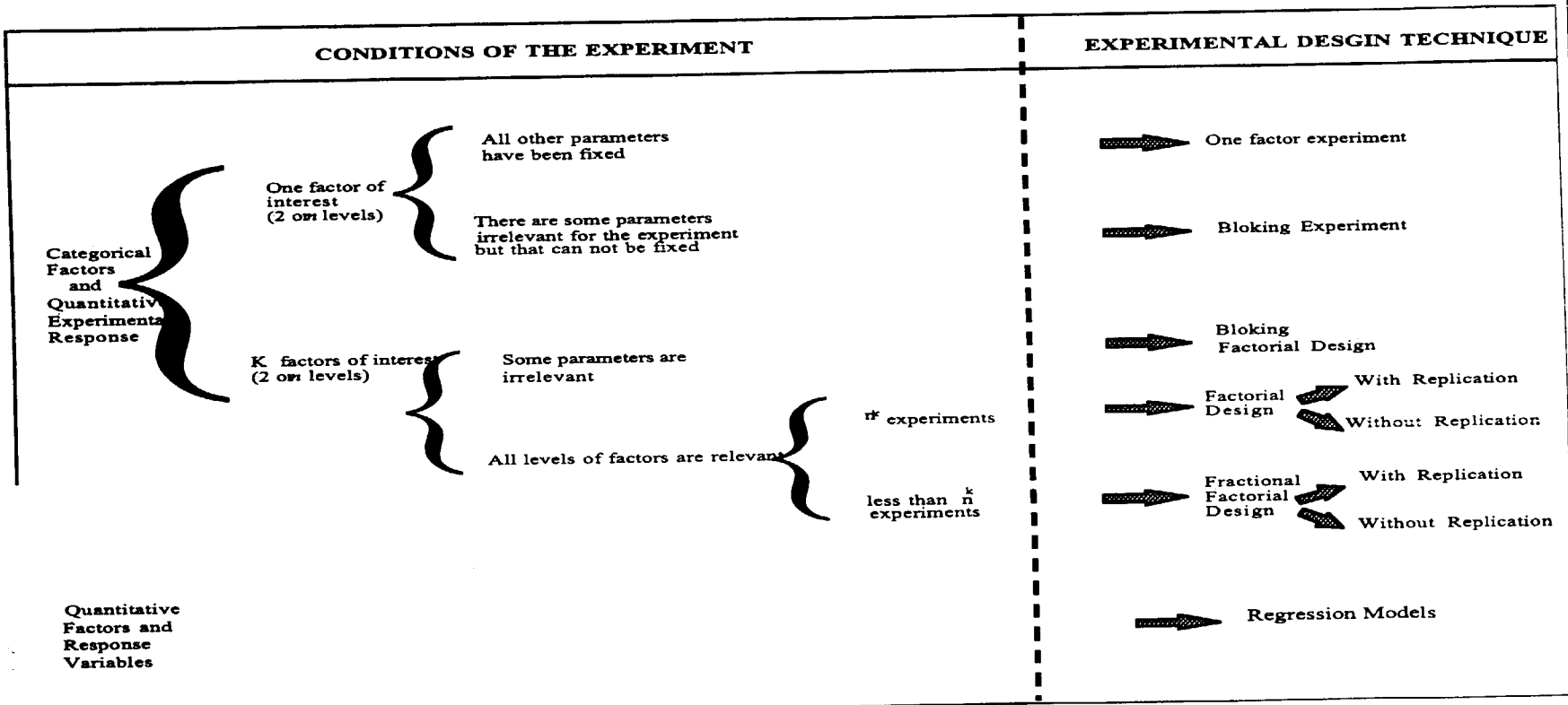
Perform Formal Experiments Using EXPERIMENTAL DESIGN Adapted to SE

SOME CONCEPTS OF EXPERIMENTAL DESIGN

- Proposed early in the 20th century by Sir Ronald Fisher
- It is routinely used by other engineering fields: Chemistry, Agriculture, Pharmaceuticals, ...
- It establishes mathematical foundations to perform experiments, choose the variables of the experiments, collect and analyse data and arrive at conclusions
- The main concept is the idea of statistical significance

SOME CONCEPTS OF EXPERIMENTAL DESIGN

Diferent Techniques for Diferent Situations



ADAPTATION OF ED CONCEPTS TO SE Terminology

Concept	Description	Application in SE
Experimental unit	Entity used to conduct the experiment	Software projects
Parameters	Characteristic (qualitative or quantitative) of the experimental unit	See
Response variable	Datum to be measured during the experimental unit	See kect table. Note there are no response variables relating to "problem". This is because response variables are data that can be measured *a posteriori*, that is, once the experiment is complete. In the case of SE, the experiment involves development (in full or in part) of a software system to which particular technologies are applied. The characteristics of the *problem* to be solved are the experiment input data, that is, they stipulate howit will be performed. As such, they are parameters and factors of the experiment. However, they are not experiment output data that can be measured and, thus, do not generate response variables.
Factor	Parameter that affects the response variable and whose impact is of interest for the study	Factors are chosen from parameters in table 3. Factors have different values during the experiment
Level	Possible values or alternatives of the factors	Values of factors in table 3
Interaction	The effect of one factor depends on the level of other	Relations between the parameters in table 3, for example problem complexity and product complexity
Replication	Repetition of each experiment to be sure of the measurement taken of the response variable	Repeatability in SE must be based on analogy, not on identity; the different experiments will consist of similar problems, similar processes, similar teams, etc.
Design	Specification of the number of experiments, selection of factors, combinations of levels of each factor for each experiment and the number of replications per experiment	The design will indicate the number of software projects, factors and their alternatives that will be used during experimentation, as well as the number of replications of the experiments, based on analogy.

ADAPTATION OF ED CONCEPTS TO SE

Experiments Parameters

PARAMETERS			
PROBLEM (User need)	PROCESSES of construction employed	PERSONS (team of developers)	PRODUCT
– Definition (poorly/well defined problem) – Need volatility (very/hardly/non volatile need) – Ease of understanding (problem well/poorly/fairly well understood by developers) – Problem complexity – Problem type (data processing, knowledge use, etc.), – Problem-solving type (procedural, heuristic, real-time problem solving, etc.) – Domain (aeronautics, insurance, etc.) – User type (expert novice, etc.)	– Maturity – Description (set of phases, activities, products, etc.) – Relationship between members (definition of interrelations between team members) – Automation (in which phases or activities tools are used) – Risks	– Number of members – Division by positions (no. of software engineers, programmers, project managers, etc.) – Years of experience of each member in development – Experience of each member in the problem type – Experience of each member in the software process applied – Background of each member (discipline of origin) – Type of relationship between members (all in the same building, same town, subcontracts, etc.)	– Type of life cycle to be followed – Software type (OO, databases, real time, expert system, etc.) – Size – Complexity – Architecture/Organization – Hardware platform – Interaction with other software – Processing conditions (batch, on-line, etc.) – Security requirements – Response-time requirements – Documentation required – Help required

ADAPTATION OF ED CONCEPTS TO SE
Response Variable

RESPONSE VARIABLES			
PROBLEM	PROCESS	PERSONS	PRODUCT
	– Schedule deviation – Budget deviation – Compliance with construction process – Products obtained (do they comply with the process stipulations?)	– Productivity – User satisfaction – usability – usefulness	– Correctness of products obtained (no. of errors, etc.) – Validity of the products (compliance with customer expectations) – Portability, Maintainability, Extendibility, Performance, Flexibility, Interoperability,...

ADAPTATION OF ED CONCEPTS TO SE
Example of Factorial Design

FACTORS:	Development Paradigm (new/00)
	Process Maturity (high/low)
	Problem Complexity (complex/simple)
RESPONSE VARIABLE:	Number of errors detected three months after deployment
ED TECHNIQUE:	Factorial Design
RESULT:	Correctness is better when the new paradigm is used

ADAPTATION OF ED CONCEPTS TO SE
Example of Factorial Design

I	A	B	C	AB	AC	BC	ABC	φ
1	-1	-1	-1	1	1	1	-1	14
1	1	-1	-1	-1	-1	1	1	22
1	-1	1	-1	-1	1	-1	1	10
1	1	1	-1	1	-1	-1	-1	34
1	-1	-1	1	1	-1	-1	1	46
1	1	-1	1	-1	1	-1	-1	58
1	-1	1	1	-1	-1	1	-1	50
1	1	1	1	1	1	1	1	86
320	80	40	160	40	16	24	9	total
40	10	5	20	5	2	3	1	total/8

$$SST = 2^2 (C_A^2 + C_B^2 + C_C^2 + C_{AB}^2 + C_{AC}^2 + C_{BC}^2 + C_{ABC}^2 =$$
$$2(102 + 52 + 202 + 52 + 22 + 32 + 12) =$$
$$800 + 200 + 3200 + 32 + 72 + 8 = 4512$$

A (Development Paradigm): 800/4512 = 18%
B (Process Maturity): 200/4512 = 4%
C (Software Complexity): 3200/4512 = 71%
AB: 200/4512 = 4%
BC: 32/4512 = 1%
AC: 72/4512 = 2%
ABC: 8/4512 = 0%

$$\text{Correctness}_{\text{paradigm = new}} = 14+46+10+50 / 4 = 30 = 40 - 10$$
$$\text{Correctness}_{\text{paradigm = oo}} = 22+58+34+860 / 4 = 50 = 40 - 10$$

Extending Enterprise and Domain Engineering Architectures to Support the Object Oriented Paradigm

Fred A. Maymir-Ducharme, PhD
Lockheed Martin, Mission Systems
fred.a.maymir-ducharme@lmco.com

1.0 BACKGROUND As the size and complexity of software systems increase and budgets decrease, the U.S. Government has realized the dire need to provide guidance to develop systems more effectively and efficiently. We can no longer afford to "reinvent the wheel" every time a new system is needed. Engineering families of systems, product lines, and exploiting commercial off-the-shelf (COTS) software and Government off-the-shelf (GOTS) software are just a few approaches to achieving better engineered systems. In addition, software intensive systems must be able to work together and exchange information. While interoperability is important for many information systems, it is essential for military systems, which must be capable of supporting lifesaving operations that may require changing a mix of forces, at a moment's notice, just about anywhere in the world.

U.S. Government has developed architectural guidance and policy to achieve the required interoperability, as well as engineering systems faster, better and cheaper. These initiatives and products only address "what" should be done. Program managers and systems engineers tasked to deliver these systems depend on technology addressing the associated "how to's." This paper addresses the technology (concepts, processes, methods and tools) used on multiple programs to effectively and efficiently engineer military systems, using various architecture guidance, policy and products. One of the major themes (i.e., lessons learned) of this paper is that there are many conflicts between the technologies associated with Object Oriented approaches and the more traditional Structured/Functional approaches. If both approaches are used by an organization, these challenges must be identified early and dealt with accordingly.

2.0 DISCIPLINED SOFTWARE ENGINEERING The way we engineer our systems is continuously changing and improving. We can no longer treat each new project as a single, new and independent development effort and not build on previous engineering efforts and experience. Instead we need to view these systems within the context of similar systems built in the past, exploiting the commonalities and engineering the appropriate variances. Additionally, we must leverage off existing reusable assets and develop new ones with reuse in mind. Reuse is an integral part of a disciplined software engineering practice, which is continuously improving its technology/asset base and processes. In order to meet today's software challenges [6] of increasing demand, complexity and size, we need to establish new ways of fusing together information about what assets exist and need to be woven into the processes used to guide our engineering activities. Various software engineering methods, processes and tools exist to help take advantage of available information about data, process, and software assets needed to make the engineering decisions governing the quality of the products that evolve as a consequence of their mechanization.

Disjoint engineering efforts (i.e., Information Engineering, Domain Engineering and Application Engineering) result in engineering process stovepipes. Each engineering level develops models representing the associated requirements. Each engineering practice designs a solution (sometimes captured by an architecture or design). And each engineering practice then implements/develops their products. The challenge is to fuse these engineering methods (and thereby their work products) to eliminate redundancies, inconsistencies and other anomalies. The goal is to define and implement a disciplined software engineering practice that assures that the work products and standards produced any phase of the lifecycle are consistent and coordinated with the work products and standards of all associated lifecycle phases. For example, data models developed during the enterprise modeling phases must feed into the appropriate domain engineering and application engineering phases; and reciprocally, provide feedback to the enterprise efforts when the data models need to be modified or extended. Applications developed individually without considering common and/or related systems in the domain result in stovepipe systems /

applications. Likewise, domains engineered without considering the broader enterprise (e.g., common data elements, business functions, the need to interoperate, etc.), can result in stovepipe domains.

Mature engineering disciplines support clear separation of routine problem solving from R&D. These disciplines have publicly-held, experience based, and formally transmitted technology bases that include product models (e.g., designs, specifications, performance ranges) and practice models (tools and techniques to apply to the product models) (See Figure 1 below). Furthermore, the qualities of products built from these models are well-understood and predictable before the products are produced.

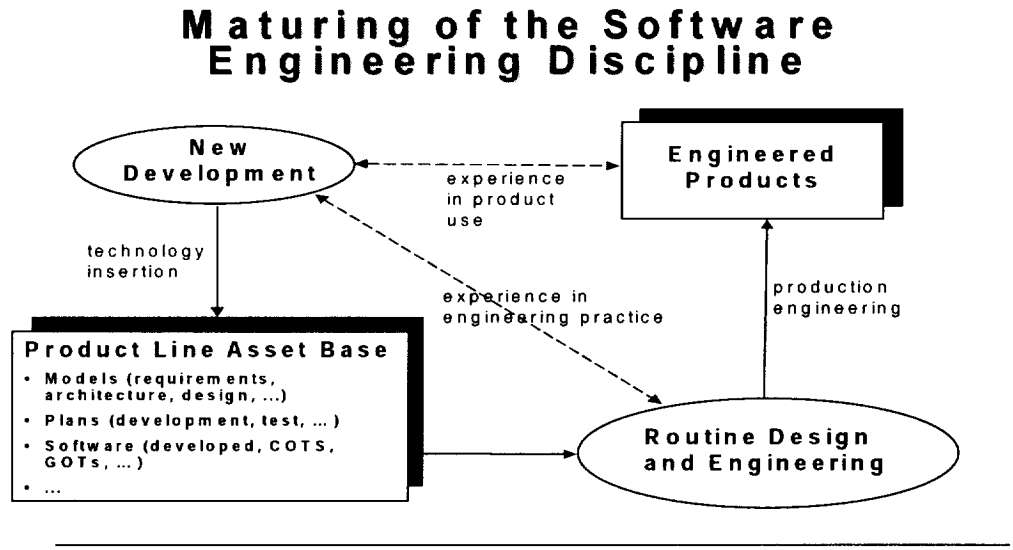


Figure 1 The Maturing of the Software Engineering Discipline

The state-of-the-practice of software engineering is not yet at this level of maturity. Instead of basing new development on a technology base of well-understood models, current software engineering practice tends to start each new application development from scratch with the specification of requirements, and moves directly into the design and implementation. By contrast, this effort's vision of a mature software engineering discipline, as illustrated in the figure above, relies on a technology base of reusable assets and clearly separates routine systems development (i.e., application engineering) from development of the domain-specific technology base (i.e., domain engineering). This separation highlights the need and significance of developing reusable corporate assets including requirements, models, architectures, processes, and components. The application engineering function can then focus on validating and using this technology base, instead of beginning with a blank sheet. In addition to creating the initial set of domain assets, domain engineering processes will continue to add and enhance the technology base according to the requirements associated with application engineering.

Under the USAF Comprehensive Approach to Reusable Defense Software (CARDS) Partnerships Program [20], LM developed and applied, the AF/CARDS Engineered Software (ACES) methodology [21,22,23] (illustrated below), an approach that combines Information Engineering with Domain Engineering and the Object Modeling Technique (OMT). The CARDS Tri-Lifecycle Software Engineering model [1,2,27] (Figure 2 below), reflects three types of engineering activities during the acquisition and life cycle development and maintenance of software intensive systems: Enterprise Engineering [2,3,26], Domain Engineering [23,24.25], and Application Engineering [1,5,7,8,17]. Due to the complexity of engineering all of the systems within the enterprise, as well as the numerous methodologies available for each engineering area, it is likely that information will be lost, regenerated, or not seen as relevant to previous or succeeding activities -- thereby causing redundant work efforts, data and function anomalies, and higher development and maintenance costs. This lack of coordination and communication across processes has been coined

"stovepipe processes" and is analogous to the systems stovepipes dilemma, where systems fail to leverage common data and the necessary interoperability. Approaching the problem with planning oversight of all three activities ensures that information flows from one activity to the next.

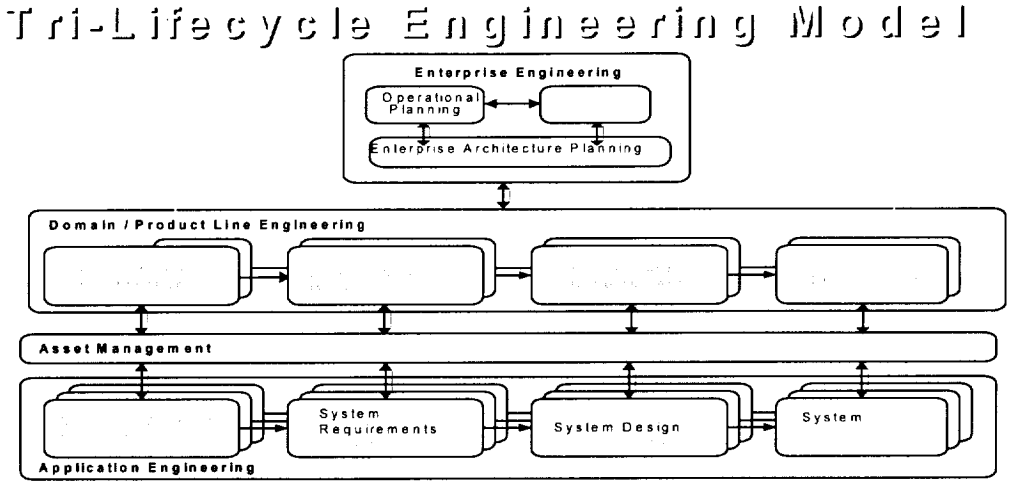


Figure 2 The CARDS Tri-Lifecycle Engineering Model

There are numerous Domain Engineering methods and processes. The primary domain analysis methods (primary because of their validation/applications on various efforts and associated publications) include: Organization Domain Modeling (ODM) [18], a well defined and comprehensive method; Domain Engineering Process (DEP) [27], an extension of object-oriented methods; the SEI Feature Oriented Domain Analysis (FODA) [2] method, considered to be the most mature DE methodology; and SPC's Synthesis [19].

3.0 ARCHITECTURE GUIDANCE U.S. Government guidance and policy such as the Command Control Communications Computers Intelligence Surveillance and Reconnaissance (C4ISR) Architecture Framework, Joint Technical Architecture (JTA), Defense Information Infrastructure (DII) Common Operating Environment (COE) and other US DoD architectural guidance are crucial to achieving interoperability, while building systems faster, better and cheaper.

The JTA [30] is the DoD's specification for interoperability between all DoD systems. Figure 3 below illustrates the relationship of the JTA to other DoD architecture guidance and initiatives. The JTA is based on the Technical Architecture Framework for Information Management (TAFIM), Adopted Information Technology Standards (AITS) – Volume 7 of the TAFIM [12]; and uses the DoD Technical Reference Model (TRM, TAFIM Vol 2) as it's structure for specifying interoperability for each major service area. The JTA defines the service areas, interfaces, and standards (JTA elements) applicable to all DoD systems, and its adoption is mandated for the management, development, and acquisition of new or improved systems throughout DoD. The JTA is complementary to and consistent with other DoD programs and initiatives aimed at the development and acquisition of effective, interoperable information systems -- including the DoD's Specification and Standards Reform, the Information Technology Management Reform Act (ITMRA); DoD C4ISR Architecture Framework, the DoD TRM; the Defense Information Infrastructure Common Operating Environment (DII COE); and Open Systems Initiative.

DOD Architecture Efforts

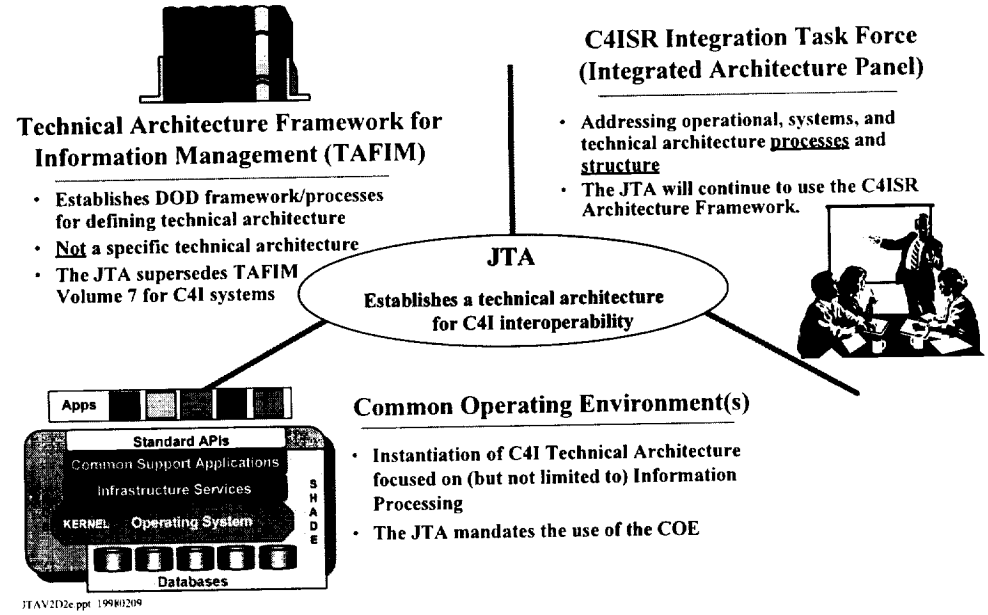


Figure 3 DoD Architecture Guidance

The DoD TRM originated from the TAFIM and was developed to show which interfaces and content needed to be identified. The TRM Working Group (TRMWG) has extended the scope of the TRM to include real-time systems (e.g., weapon systems) and is coordinated with the JTA. As figure 3 indicates, the JTA is also very closely coupled with the DII COE [32] and the C4ISR Architecture Framework [31]. The DII COE is the DoD's implementation of a technical architecture supporting interoperability, supplemented by various common services / utilities to maximize reuse across multiple systems. And as the figure below indicates, the JTA is one of the three architectures defined by the C4ISR Architecture Framework.

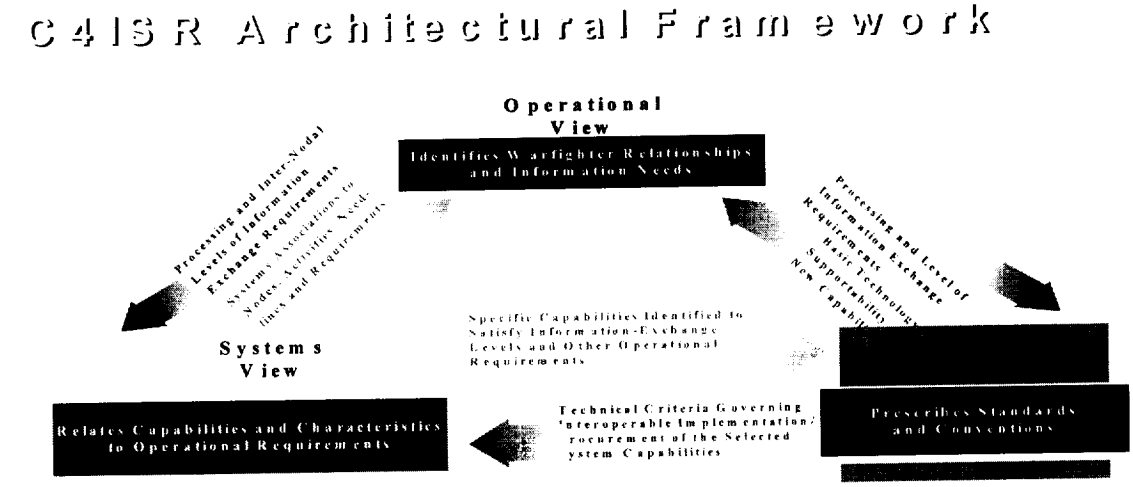


Figure 4 DoD C4ISR Architecture Framework

4.0 LESSONS LEARNED Lockheed Martin (LM) worked with the USAF to replace existing transportation information systems. These systems were designed as stand-alone applications serving individual offices or functions. The resulting system gaps and overlaps, and the concomitant data and process redundancy and inconsistency, have caused problems for both information users and systems maintainers. USAF's goal is to reduce development and maintenance costs while enhancing support to the warfighter. Its objectives are to develop a unified transportation system and environment -- consisting of a corporate database, corporate applications, common functionality, and a corporate network. The strategy for reaching these objectives is to introduce a reuse-based approach to application systems development. The approach is to replace stovepipe information systems with a set of integrated applications that cut across organizational and functional lines and to implement a virtual corporate database. The corporate database will appear to the user to be integrated and monolithic but will actually be composed of physically distributed, heterogeneous databases and - for the foreseeable future - legacy USAF and DoD systems.

The USAF employed the Zachman Framework to guide its Information Systems Architecture development. Within this framework, USAF addressed its enterprise-wide data integration objectives by applying Steven Spewak's Enterprise Architecture Planning (EAP) process (an Information Engineering (IE) technique). The product, a high-level Transportation System Master Plan, includes a Mission Analysis, Information Architecture, Application Architecture, and Implementation Plan.

ACES was based on the CARDS Tri-Lifecycle Engineering Model, which extended the DARPA Software Technology for Adaptable, Reliable Systems (STARS) Dual Lifecycle Model (i.e., Domain Engineering) to include Information/Enterprise Engineering. The complete ACES methodology addresses Enterprise Engineering (e.g., Spewak's EAP) [3], Object-Oriented (OO) Domain Engineering, and OO Applications Engineering (using Rumbaugh's OMT) [1]. The transition from Enterprise Engineering to Domain Engineering uses IE-based affinity analysis between data entities and business processes to identify and scope candidate domains. It then uses an OO approach to analyze inter-domain relationships in terms of service requests. Within each domain of focus, ACES uses FODA to identify and categorize reuse opportunities, and OMT to develop reusable business objects that satisfy semantic information integration and synthesis requirements. Application Engineering consists of matching specific user requirements to business objects and developing the necessary application-specific objects.

There were many lessons learned throughout this effort. Transitioning from the very functional (sometimes referred to as "structured") information/enterprise engineering methods to an OO solution incurred several challenges. Applying affinity analyses and multi-domain modeling techniques over the enterprise information element lifecycles to scope the domains and hence group the service objects proved to be key in this transition. The fundamental differences between structured and OO approaches must be considered in the many translations and transitions across the various methods and workproducts within the Tri-Lifecycle. The Data Access Layer within the framework in Figure 4 below was necessary to deconflict data access between the structured legacy code and the new OO code. The figure below summarizes the integration and application of DoD architectural guidance / products with the associated architecture technology. Lessons learned will be discussed during the panel session. Additional lessons learned in applying the ACES methodology, based on the CARDS Tri-Lifecycle Engineering Model above are discussed in the references listed below. The figure below illustrates the integration of both, the technology (e.g., EAP, ACES, OMT) and the DoD guidance/products (e.g., C4ISR Architecture Framework, JTA and COE) used to reengineer the USAF's Defense Transportation System.

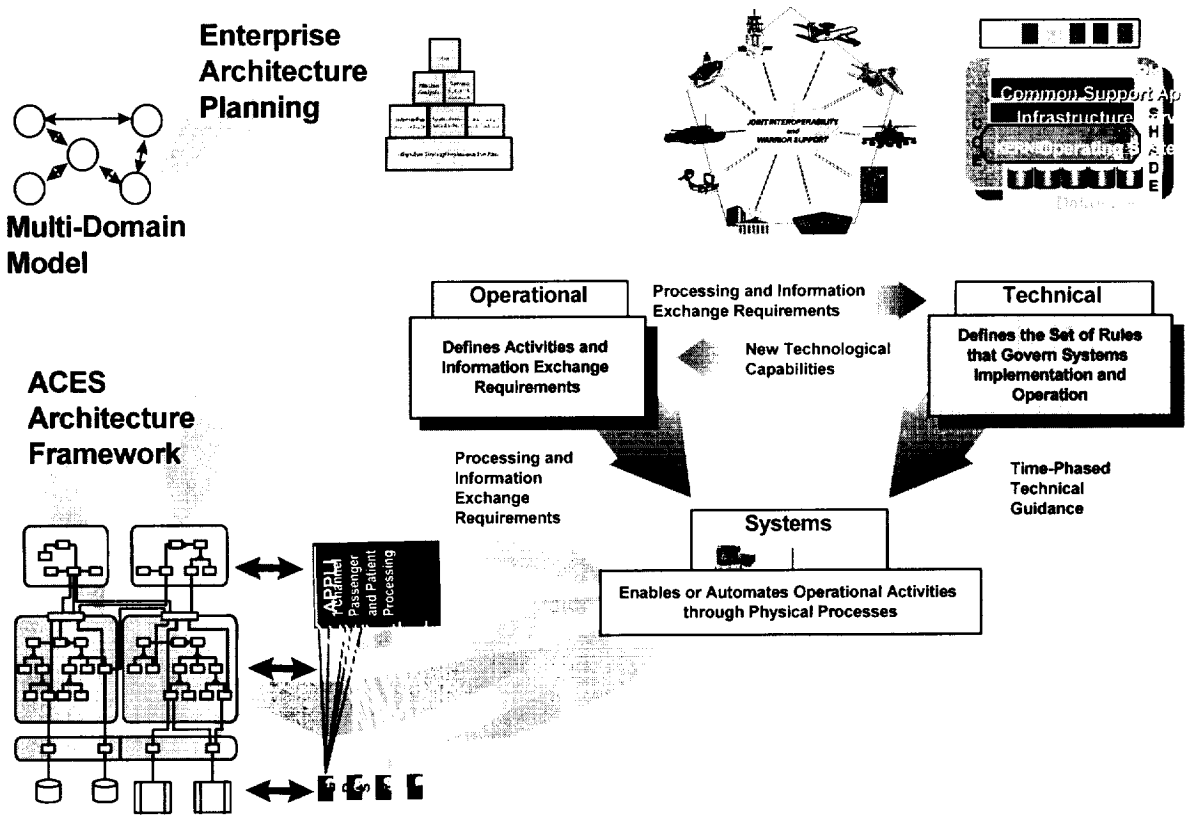


Figure 5 ACES'Integration of Architecture Guidance, Policy & Technology

5.0 ACKNOWLEDGMENTS Special thanks and credit are due to the teams that worked with me on the development and application of these technologies, as well as the team supporting the development of the DoD JTA (2.0). These include Jim Fulton, Mike Webb, Robin Burdick, Frank Svoboda, Roger Whitehead, David Weisman, Lucy Haddad, Nancy Solderitsch, Paul Kogut, Wil Berrios, Russ Richards, Olimpia Velez, Jim DeGoey, Mark Dowson and many others.

6.0 REFERENCES

1. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. Object-Oriented Modeling and Design, Prentice-Hall, 1991.
2. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study, CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute, November 1990.
3. S. Spewak. Enterprise Architecture Planning, John Wiley & Sons, 1992
4. J. Zachman. "A Framework for Information Systems Architecture," IBM Systems Journal, Vol. 26, No. 3, 1987.
5. W. Royce, "Managing the Development of Large Software Systems: Concepts & Techniques" 1970
6. OUSD/AT "Defense Report of the Defense Science Board Task Force on Military Software," 1987
7. B. Boehm, "A Spiral Model of Software Development an Enhancement," 1988.
8. W. Royce, "TRW's Ada Process Model for Incremental Development of Large Software Systems," 1990.
9. MIL-STD-498 "Software Development and Documentation" 1994
10. EIA/IEEE J-STD-016 "Software Life-Cycle Processes" 1995
11. ISO/IEC STD 12207 "IT – Software Life-Cycle Processes" 1995
12. DoD, "Technical Architecture Framework for Information Management (TAFIM)" Version 2.0, Defense Information Systems Agency, Center for Architecture, June 1994
13. The DoD Enterprise Model, Volume I: Strategic Activity and Data Models, Office of the Secretary of Defense, ASD (C3I), January 1994.

14. The DoD Enterprise Model, Volume II: Using the DoD Enterprise Model, A Strategic View of Change in DoD, A White Paper, Office of the Secretary of Defense, ASD (C3I), January 1994.
15. Information Management Program, DoD Directive 8000.1, October 1992.
16. DoD Data Administration, DoD Directive 8320.1, September 1991.
17. IEEE Standard for Developing Software Life Cycle Processes, IEEE Computer Society, IEEE STD 1074-1991, January 1992.
18. Simos, M., "ARPA STARS Organization Domain Modeling (ODM) Guidebook Version 1.0" March 1995
19. "Synthesis, A Reuse-Based Software Development Methodology, Process Guide, Version 1.0," Software Productivity Consortium, October 1992.
20. Maymir-Ducharme, F.A., Weisman, D. "A.F./CARDS Technology Transition Program: Reuse Partnerships," proceedings of the Reuse '95 Workshop, August 1995.
21. Maymir-Ducharme, F.A., "Variant Domain Engineering Approaches," proceedings of the Workshop on Institutionalizing Software Reuse WISR'95, July 1995.
22. Maymir-Ducharme, F.A., Svoboda, F. "Translating Enterprise Models into Domain Engineering Workproducts," Proceedings of the Reuse '96 Workshop, August 1996.
23. Maymir-Ducharme, F.A., (WG Chair). "Opportunistic, Systematic and Optimized Domain Engineering Approaches" Proceedings of the Reuse '96 Workshop, August 1996.
24. Maymir-Ducharme, F.A., "Product Lines, Just One of Many Domain Engineering Approaches," Proceedings of the NASA Software Reuse Workshop, sponsored by GMU and NASA SORT Program, October 1997,
25. Maymir-Ducharme, F.A., "A Product Line Business Model," Proceedings of ARES'96 (Architectural Reasoning for Embedded Software), sponsored by ESPRIT IV project no. 20.477, Las Navas, Spain, 18-20 Nov. 1996.
26. Martin, James, "Information Engineering : A Trilogy," Prentice Hall, Inc., Englewood Cliffs, NJ 1989.
27. Defense Information Systems Agency (DISA), "Domain Engineering Process (Version 2)" 28 April 1995.
28. Combined Communications Electronics Board (CCEB), "Combined Interoperability Technical Architecture (CITA) Rationale and Development Framework (Ver. 0.2) March, 1998.
29. CCEB, "Combined Interoperability Technical Architecture (CITA), Ver. 0.1," March 1998.
30. DISA, "DOD Joint Technical Architecture (JTA)," http://www-jta.itsi.disa.mil/
31. OSD/C3I "C4ISR Architecture Framework," http://www.cisa.osd.mil/organization/architectures/
32. DISA, "Defense Information Infrastructure (DII) Common Operating Environment (COE)," http://spider.osfl.disa.mil/dii/

Session 3: Inspections

National Software Quality Experiment: A Lesson in Measurement: 1992 - 1997
D. O'Neill, Independent Consultant

Principles of Successful Software Inspections
D. Beeson, Ki Solutions Consulting, and T. Olson, World-Class Quality

Capture-Recapture - Models, Methods, and the Reality
J. Ekros and A. Subotic, Linkoeping University

NATIONAL SOFTWARE QUALITY EXPERIMENT
A LESSON IN MEASUREMENT
1992-1997

7-61

KEY WORDS
Analysis Bins
Common problems
Core samples
Defect types
Experiment participants
Software Inspection Lab
Software process maturity level
Standard of excellence
Return on investment

PROLOGUE

The nation's prosperity is dependent on software. The nation's software industry is slipping, and it is slipping behind other countries. The National Software Quality Experiment is riveting attention on software product quality and revealing the patterns of neglect in the nation's software infrastructure.

ABSTRACT

In 1992 the DOD Software Technology Strategy set the objective to reduce software problem rates by a factor of ten by the year 2000. The National Software Quality Experiment is being conducted[1] to benchmark the state of software product quality and to measure progress towards the national objective.

The National Software Quality Experiment is a mechanism for obtaining core samples of software product quality. A micro-level national database of product quality is being populated by a continuous stream of samples from industry, government, and military services. This national database provides the means to benchmark and measure progress towards the national software quality objective and contains data from 1992 through 1997.

The centerpiece of the experiment is the Software Inspection Lab where data collection procedures, product checklists, and participant behaviors are packaged for operational project use. The uniform application of the experiment and the collection of consistent measurements are guaranteed through rigorous training of each participant. Thousands of participants from dozens of organizations are populating the experiment database with thousands of defects of all types along with pertinent information needed to pinpoint their root causes.

To fully understand the findings of the National Software Quality Experiment, the measurements taken in the lab and the derived metrics are organized along several dimensions including year, software process maturity level, organization type, product type, programming language, and industry type. These dimensions provide a framework for populating an interesting set of analysis bins with appropriate core samples of software product quality.

[1] The National Software Quality Experiment is an entrepreneurial activity.

EXPERIMENT MOTIVATION AND ORGANIZATION

Overview

Participants are attracted to the experiment as a place where they can calibrate their software quality against appropriately selected industry core samples. Here they can jump-start the organization's quality measurement program on the shoulders of uniform Software Inspection Lab procedures. These procedures are operationally packaged for project use and include well defined processes, industrial strength product checklists, participant roles and behaviors, and standard forms and reports.

The National Software Quality Experiment provides the framework to pose important quality questions. Its micro-level national quality database provides the measurements to answer them. Similarly, the extent of certain common risks can be quantified. As a participant in the experiment, an organization can characterize the effectiveness of its software quality process. At the industry level, progress towards the national software quality objective can be can be benchmarked.

Participants in the experiment benefit in several ways. They are able to characterize the maturity of their software quality process. With this understanding, they are able to establish goals for improving the process and to set priorities for immediate action. Beyond that, these organizations are able to promote a vision for excellence in their software products and to calibrate their progress towards the national software quality goal.

Motivation

The Department of Defense Software Technology Strategy was drafted for the Director of Defense Research and Engineering in December 1991 [DOD STS 91]. Three important national objectives were established to be achieved by the year 2000:

1. Reduce equivalent software life-cycle costs by a factor of two
2. Reduce software problem rates by a factor of ten
3. Achieve new levels of mission capability and interoperability via software

Every software organization should treat the national objective to improve software product quality by a factor of ten as a wake-up call. Are organizations planning to reduce software problem rates by a factor of ten? Do they know what these rates are now?

Measurement Best Practice

Although measurement is needed to derive effective policy governing acquisition, development, and operations, there is not yet an industry consensus on the wisdom of creating a national database for software engineering. The issue centers on the use of the data, not on its collection. The worry is that the industry is not ready to use the database appropriately. Clearly the industry can learn to use the database appropriately once it exists. If there are national goals set for software engineering, there must also be a national measurement program and database to track progress and refine goals. Carnegie Mellon University's Software Engineering Institute produced "A Concept Study for a National Software Engineering Database" in July 1992 [Van Verth 92]. The study points out that there are many users for such a database, but few suppliers. The study offers the following observations and advice on establishing a national database:

1. Wide variance may exist in the collection process
2. Common data definitions are needed
3. Goals and questions should precede data collection
4. Confidentiality of the data must be protected

In designing the experiment, it is recognized that the prescription for achieving lasting value in measurement depends on the successful integration of measurement concepts, operationally defined and packaged processes, effective technology transition including the training of participants and the dissemination of results, and hands-on oversight of the experiment. The prescription for lasting value in measurement revolves around four driving measurement concepts. First, measurement must be aligned with business needs. Second, metrics must be carefully pinpointed and rigorously defined. Third, measurement activities must be built into the normal operation of the organization. Finally, extraordinary steps must be applied to obtain consistency and uniformity in data collection.

Finally, Dr. Vic Basili of the University Maryland provides the following additional guidelines [Wallace 97]:

1. Establish the goals of the data collection
2. Develop a list of questions of interest
3. Establish data categories
4. Design and test the data collection form
5. Analyze data

Nature and Role of Experiment

In the practice of software engineering, managers are guided more by myth than by measurement. The experiment provides the framework for measuring critical aspects of software product quality practice. The framework supplies the ingredients needed to install a uniform and consistent measurement methodology. These are described in the Software Inspections Mechanism. The predictability of the measurements taken in conducting the experiment provides the basis for assessing the validity of a hypothesis. This is discussed in Experiment Results. Some of the questions asked and answered in the experiment are:

1. To what extent is there a continuing stream of requirements changes?
2. What are the leading types of errors?
3. Are errors traced to people or process?
4. Is a standard development process followed?
5. To what extent are wrong software functions being developed?
6. To what extent are there shortfalls in real time performance?
7. Is gold plating a problem?

Software inspections are an essential ingredient in fact-based software management. They utilize a reasoning process for conducting a fine-grained, deep-probing evaluation. When combined with automation-based quick-look evaluations, the best balance between efficiency and insight can be obtained. Once installed in the organization, the software inspection process yields core samples of software product quality. These can be used to benchmark problem rates by defect type among major product areas within the organization. With the benchmark measurements in place, the software inspections process provides a stable, uniform, and persistent mechanism for measuring improvement progress toward the software product goals of the organization.

SOFTWARE INSPECTIONS MECHANISM

Setting the Standard of Excellence

Software products reveal the standard of excellence in software engineering applied in their production. In improving software product quality, an industrial strength standard of excellence must be set, and the software operations within the organization must be disciplined to meet that standard. This is done by measuring actual practice using the strongly preferred indicators from the national standard of excellence spanning completeness, correctness, style, rules of construction, and multiple views.

Completeness
Completeness is based on traceability among the requirements, specification, design, code, and test artifacts. Completeness analysis reveals what predecessor artifact sections have not been satisfied as well as the inclusion of extra fragments.

Correctness
Correctness is based on reasoning about programs through the use of informal verification and correctness questions derived from the prime constructs of structured programming and their composite use in proper programs. Input domain and output range are analyzed for all legal values and all possible values. Adherence to project specific disciplined data structures is analyzed.

Style
Style is based on project specified style guidance based on block structured templates. Semantics of the design and code are analyzed for correspondence to the semantics used in the requirements, specifications, and design. Naming conventions are checked for consistency of use; and commentary, alignment, upper/lower case, and highlighting use are checked.

Rules of Construction
Rules of construction are based on the software architecture and its specific protocols, templates, and conventions used to carry it out. For example, these include interprocess communication protocols, tasking and concurrent operations, program unit construction, and data representation.

Multiple views
Multiple views are based on the various perspectives required to be reflected in the product. During execution many factors must operate as intended including initialization, timing of processes, memory use, input and output, and finite word effects. In generating the software, packaging considerations must be coordinated including program unit construction, program generation process, and target operations. Product construction disciplines of systematic design and structured programming must be followed as well as interfaces with the user, operating system, and physical hardware.

EXPERIMENT RESULTS

Experiment Participants
The participants of the National Software Quality Experiment have been trained in the Software Inspections Course and Lab. Experiment results are drawn from these Inspection Lab sessions. The participating organizations span government, DOD industry, and commercial sectors and represent a wide range of application domains.

- Accounting, personnel, administration
- Administrative and management decision support
- Aircraft jet engine diagnostics, logistics, and maintenance
- Artillery fire control system
- Avionics flight on-board control
- Control devices for avionics applications
- Credit card application
- Department of State embassy support
- Electronic commerce
- Electronic warfare
- FAA communications
- Factory line support
- Financial services
- Global positioning system user sets
- Insurance and medical information
- International banking
- Joint Chiefs of Staff support
- Medical information system
- Naval surface weapons system
- Pre and post flight space application
- Telecommunications

Results Summary

Ralph Waldo Emerson said, "The years teach us things the days never knew". The National Software Quality Experiment has been accumulating a steady stream of core samples for its micro-level national database. These results provide a benchmark of software product quality measurements useful in assessing progress towards the national software quality objective for the year 2000. These results are highlighted below in the discussion of the common problems pinpointed, defect category and severity data summary, Inspection Lab operations, the predictability of certain measurements, and the ranking of defect types.

Common Problems

Analysis of the issues raised in the experiment to date has revealed common problems that reoccur from session to session. Typical organizations which desire to reduce their software problem rates should focus on preventing the following types of defects:

1. Software product source code components are not traced to requirements.
   - *As a result, the software product is not under intellectual control, verification procedures are imprecise, and changes cannot be managed.*
2. Software engineering practices for systematic design and structured programming are applied without sufficient rigor and discipline.
   - *As a result, high defect rates are experienced in logic, data, interfaces, and functionality.*
3. Software product designs and source code are recorded in an ad hoc style.
   - *As a result, the understandability, adaptability, and maintainability of the software product are directly impacted.*
4. The rules of construction for the application domain are not clearly stated, understood, and applied.
   - *As a result, common patterns and templates are not exploited in preparation for later reuse.*

Defect Category and Severity

The defect severity metric revealed that 14.27% of all defects were major, and 85.73% minor. Defect category distinguishes missing, wrong, and extra. For major defects, 7.44% were missing, 5.95% wrong, and .88% extra. For minor defects, 49.76% were missing, 27.63% wrong, and 8.32% extra.

Defect Severity and Category Summary

	Missing	Wrong	Extra	Total
Major	7.44	5.95	.88	14.27
Minor	49.76	27.63	8.32	85.73
Total	57.20	33.60	9.20	100.00

Inspection Lab Operations

Through 1997 there have been 112 Inspection Labs in which 2317 participants were trained and conducted inspection sessions. A total of 788,459 source lines of code have received strict and close examination using the packaged procedures of the lab. There have been 142,306 minutes of preparation effort and 52,196 minutes of conduct time expended to detect 11,375 defects.

Of these 11,375 defects, 1854 were classified as major, and 9521 as minor. A major defect effects execution; a minor defect does not. It required 12.51 minutes of preparation effort on the average to detect a defect. To detect a major defect required 76.76 minutes of preparation effort on the average. On the average, .906 thousand source lines of code were examined each inspection conduct hour. There were 2.35 major defects detected in each thousand lines, and 12.08 minor defects. There were 4.91 defects detected per session with a return on investment of 4.48.

INSPECTION LAB OPERATIONS

Sessions	Prep Effort	Conduct Time	Major Defects	Minor Defects	Size in Lines
2317	142,306	52,196	1854	9521	788,459

Metrics:

1.	12.51	minutes of preparation effort per defect
2.	76.76	minutes of preparation effort per major defect
3.	2.35	major defects per KSLOC
4.	12.08	minor defects per KSLOC
5.	906	lines per conduct hour
6.	4.91	Defects per session
7.	4.48	Return on Investment

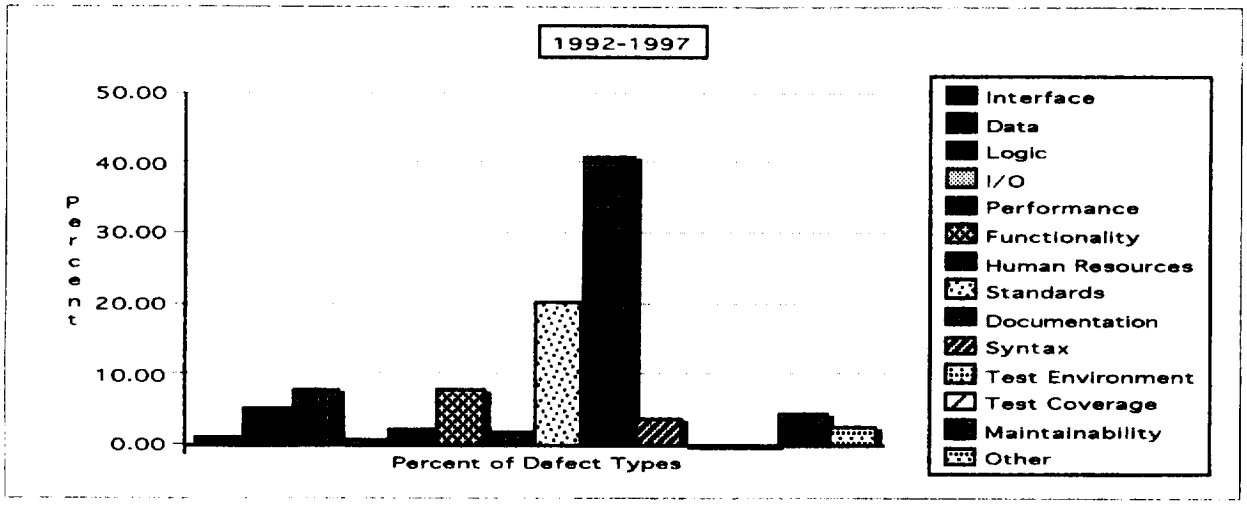
Questions Answered in the Lab

The micro-level national database on software product quality can be used to answer important software engineering questions. When appropriately selected core samples are accumulated in the Report Summary Form and the probability of occurrence is computed for each defect type, defect severity, and defect category, these probabilities can be used to construct answers to questions. Five of Boehm's top ten risks are answered below using the 1992-1997 data from the experiment:

Defect Type Ranking

The foremost defect types that accounted for 90% of all defects detected are:

Documentation	40.86%	error in guidance documentation
Standards	20.39%	error in compliance with product standards
Functionality	7.95%	error in stating or meeting intended
Logic	7.86%	error revealed through informal correctness questions function
Data	5.36%	error in data definition, initial value setting, or use
Maintainability	4.73%	error in good practice impacting the supportability and evolution of the software product
Syntax	4.02%	error in language defined syntax compliance



To what extent were the wrong software functions being developed?

Functionality errors accounted for 7.95% of all errors.

To what extent were the wrong user interfaces developed?

Interface errors accounted for 1.05% of all errors.

Human Factors accounted for 1.79% of all errors.

To what extent was there gold plating?

9.20% of all errors were classified as extra.

To what extent was there a continuing stream of requirements changes?

Documentation errors accounted for 40.86% of all errors.

To what extent was there a shortfall in real time performance?

Performance errors accounted for 2.39% of all errors.

Questions Not Yet Answered

It is useful to keep in mind that defects detected do not equal defects inserted. Defects may go undetected and leak into downstream activities. Consequently there is interest in defect leakage and ways to measure and reason about it. The Software Inspection Lab includes a mechanism to collect data on defect leakage and to reason about the results. This reasoning process crosses over into defect prevention.

Defect leakage was introduced into the National Software Quality Experiment in 1995, and the data on this is starting to build up. The defect leakage data needs to populate each analysis bin in sufficient quantity before these results are usable. With this data it will be possible to conduct special studies on defect leakage to augment the core analyses done continuously.

Questions asked but not yet answered include:

1. To what extent is defect leakage occurring?
2. What is the frequency distribution of defect types that leak?

The mechanism used to gather defect leakage involves identifying the life cycle activity for each software inspection and the defect origin for each defect. Each software inspection is considered an exit criteria for a software product engineering activity. Each defect is characterized by category, severity, type, ... and defect origin. Defect origin is the software product engineering activity where the defect was inserted. Where defect origin does not match the software product engineering activity for which this inspection serves as an exit criteria, defect leakage has occurred.

Measurement Results By Analysis Bin

The findings of the National Software Quality Experiment are organized along several dimensions which provide a framework for populating an interesting set of analysis bins with appropriate core samples of software product quality. The analysis bins are used to organize the findings into collections of data that reveal distinctions and may suggest interesting trends.The types of bins selected are year, software process maturity (level 1,2,3), organization type (commercial, DOD industry, government), product type (embedded, organic), programming language (modern, old style), and industry type (defense, financial, manufacturing, medical, telecommunication, transportation). As data for each year is collected, the overall results become more interesting, and the population of analysis bins becomes more robust.

Return On Investment

Managers are interested in knowing the return on investment to be derived from software process improvement actions. The Software Inspections Process gathers the data needed to determine this.

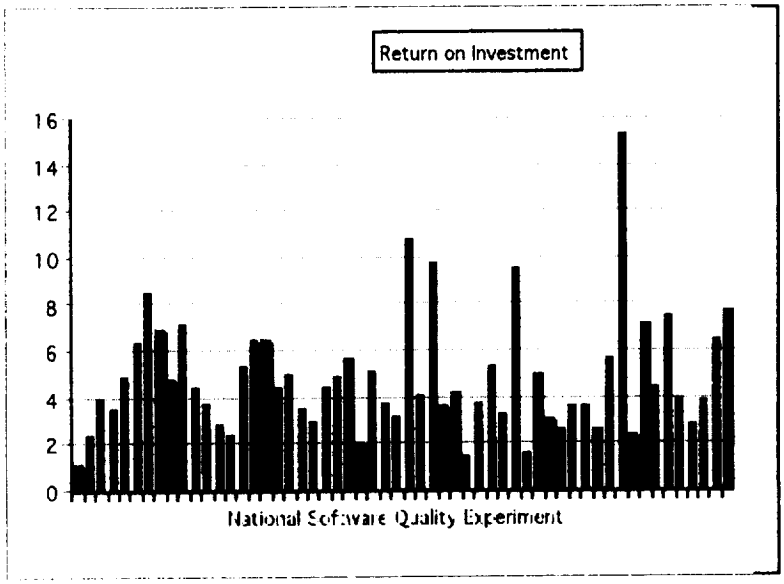
The defined measurements collected in the Software Inspections Lab may be combined in complex ways to form this derived metric. The Return on Investment for Software Inspections is defined as:

Savings/Cost , where:

Savings=(Major Defects * 9) + Minor Defects

Cost= (Minutes of Preparation Effort + (Minutes of Conduct Time * 4))/60

This model for Return on Investment bases the savings on the cost avoidance associated with detecting and correcting defects earlier rather than later in the product evolution cycle. A Major Defect that leaks into later phases may cost ten hours to detect and correct. Ten hours to fix later minus one hour to fix now results in the constant nine (9) applied to Major Defects. A Minor Defect may cost two hours to fix later minus one hour to fix now resulting in a constant of one (1) applied to Minor Defects. To convert the Minutes of Conduct Time to effort, the average number of participants (4) is applied. The constant 60 minutes is applied to convert minutes to hours.



The graph showing the Return on Investment for each organization participating in the National Software Quality Experiment suggests that the Return on Investment for software inspections ranges from 4:1 to 8:1. For every dollar spent on software inspections, the organization can expect to avoid 4-8 dollars on higher cost rework.

CONCLUSION

Closing Observations

In closing it needs to be stated that the data does not suggest progress towards the Year 2000 goal to reduce software problems by a factor of ten. Hunting for defects in software is a target rich opportunity. The harder the project looks for errors, the more it finds. The way to look harder is to reduce the volume of product inspected in each session.

The data suggests that increased software process maturity results in increased defect detection, with the result perhaps being lower defect leakage into the field. At level 1 the project lacks a shared vision for a standard of excellence for software engineering products. At level 2 attention is paid to establishing a standard of expectation, a standard of excellence, and so more defects are identified. At level 3 the standard is set and the well defined, fined grained processes for software product engineering are in place and in practice with software inspections operating as the exit criteria for each activity of the life cycle.

The data also suggests that defect density decreases with program size. As stated earlier, all programs contain a beginning, an end, and a context for operation within the larger system. Starting, finishing, and fitting in are all more error prone than the body of the program which gives it size.

In addition the data suggests that the organization's neglect of its software process exceeds the poor workmanship of individual programmers as the source of errors. Documentation and standards defect types account for nearly two-thirds of all defects, and these are the responsibility of the organization and its process.

Software products are not well connected to the requirements or business case that inspired their creation. Much of the documentation type defect detection results from the lack of traceability from the code to the design to the specification to the requirements.

Field Measurement Lessons

In conducting the National Software Quality Experiment, valuable lessons in field measurement are being learned. These lessons are forming the prescription for obtaining lasting value in measurement:

1. Measurement must be aligned with business and performance needs. These activities must be built into the normal operation of the organization. To do this, the goals to be met and questions to be answered in management, engineering, and operations must precede the collection of data.
2. Metrics must be carefully pinpointed and rigorously defined. Extraordinary steps must be applied to obtain consistency and uniformity. Without a well defined process for data collection and analysis, the variance in the measurement process itself impacts the accuracy of results.
3. Attention must be paid to the confidentiality of results. The opportunity for improvement is increased when the measured results are made more widely available. However, individuals and groups naturally resist having their shortcomings made public. If ignored, this resistance will defeat the measurement program. The organization must strike a balance between public and private data.

Next Steps

The National Software Quality Experiment is a demonstrated mechanism for collecting uniform and consistent measurements of software product quality. It provides the vantage point for software product quality and the field experience in measurement needed to jump start the practice of fact-based software management.

As the centerpiece of the experiment, the Software Inspection Labs have been installed in software factories around the country. The National Experiment collects, organizes, and packages core samples of software product quality. These measurements are increasing the understanding of the state of the practice and how to measure it.

The usefulness and success of the National Software Quality Experiment depends on sustaining a continuous stream of core samples. Organizations from industry, government, and the military are invited to participate and enrich this national database resource.

BIBLIOGRAPHY

[DOD STS 91]	Department of Defense Software Technology Strategy, draft prepared for the Director of Defense Research and Engineering [DDR&E], December 1991
[Ebenau 94]	Ebenau, Robert G. and Susan H. Strauss, "Software Inspection Process", McGraw-Hill, Inc., 1994
[Fagan 76]	Fagan, M., "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, 15, 3, 1976, 182-211
[Florac 92]	Florac, William B., "Software Quality Measurement: A Framework for Counting Problems and Defects", CMU/SEI-92-TR-22, September 1992
[Freedman 90]	Freedman, D.P., G.M. Weinberg, "Handbook of Walkthroughs, Inspections, and Technical Reviews", Dorset House Publishing Co., Inc., 1990
[Gilb 93]	Gilb, Tom and Dorothy Graham, "Software Inspection", Addison Wesley Longman Limited, 1993
[Linger 79]	Linger, R.C., H.D. Mills, B.I. Witt, "Structured Programming: Theory and Practice", Addison-Wesley Publishing Company, Inc., 1979
[Humphrey 89]	Humphrey, Watts S., "Managing the Software Process", Addison-Wesley Publishing Company, Inc., 1989
[O'Neill 88]	O'Neill, Don and Albert L. Ingram, "Software Inspections Tutorial", Software Engineering Institute Technical Review 1988
[O'Neill 89]	O'Neill, Don, "Software Inspections Course and Lab", Training Offering for Practitioners, Software Engineering Institute, 1989
[O'Neill 92]	O'Neill, Don, "Software Inspections: More Than a Hunt for Errors", CrossTalk, Issue 30, January 1992
[O'Neill 94]	O'Neill, Don, "National Software Quality Experiment", International Conference on Software Quality, Washington DC, 1994
[O'Neill 95,96]	O'Neill, Don, "National Software Quality Experiment: Results 1992-1995", Software Technology Conference, Salt lake City, 1995 and 1996
[O'Neill 97]	O'Neill, Don, "Issues in Software Inspection", IEEE Software, Vol .14 No 1., January 1997
[O'Neill 97]	O'Neill, Don, "Setting Up a Software Inspection Program", CrossTalk, The Journal of Defense Software Engineering, Vol. 10 No. 2, February 1997
[O'Neill 97]	O'Neill, Don, "National Software Quality Experiment: A Lesson in Measurement 1992-1996", Quality Week Conference, San Francisco, May 1997 and Quality Week Europe Conference, Brussels, November 1997
[O'Neill 98]	O'Neill, Don, "Software Inspections and the Year 2000 Problem", CrossTalk, The Journal of Defense Software Engineering, Vol. 11 No. 1, January 1998
[Paulk 95]	Paulk, Mark C., "The Capability Maturity Model: Guidelines for Improving the Software Process", Addison-Wesley Publishing Company, 1995
[Van Verth 92]	Van Verth, Patricia B., "A Concept Study for a National Software Engineering Database", CMU/SEI-92-TR-23, July 1992
[Wallace 97]	Wallace, Dolores R., Laura M. Ippolito, and Herbert Hecht, "Error, Fault, and Failure Data Collection and Analysis", http://hissa.ncsl.nist.gov, Quality Week, San Francisco, May 1997

AUTHOR: Don O'Neill

Don O'Neill is a seasoned software engineering manager and technologist currently serving as an independent consultant. Following his twenty-seven year career with IBM's Federal Systems Division, Mr. O'Neill completed a three year residency at Carnegie Mellon University's Software Engineering Institute (SEI) under IBM's Technical Academic Career Program. There he developed a blueprint for charting software engineering evolution in the organization including the training architecture and change management strategy needed to transition skills into practice.

As an independent consultant, Mr. O'Neill conducts defined programs for managing strategic software improvement. These include implementing an organizational Software Inspections Process, implementing Software Risk Management, and conducting Global Software Competitiveness Assessments. Each of these programs includes the necessary practitioner and management training.

In his IBM career, Mr. O'Neill completed assignments in management, technical performance, and marketing in a broad range of applications including space systems, submarine systems, military command and control systems, communications systems, and management decision support systems. He was awarded IBM's Outstanding Contribution Award three times:

1. Software Development Manager for the Global Positioning Ground Segment (500,000 source lines of code) and a team of 70 software engineers within a $150M fixed price program.
2. Manager of the FSD Software Engineering Department responsible for the origination of division software engineering strategies, the preparation of software management and engineering practices, and the coordination of these practices throughout the division's software practitioners and managers.
3. Manager of Data Processing for the Trident Submarine Command and Control System Engineering and Integration Project responsible for architecture selections and software development planning (1.2M source lines of code).

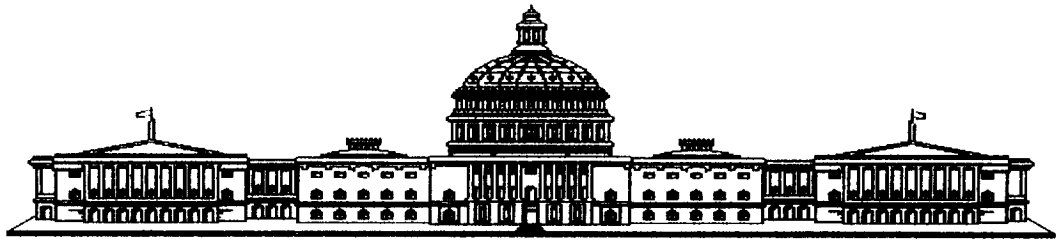
Mr. O'Neill served on the Executive Board of the IEEE Software Engineering Technical Committee and as a Distinguished Visitor of the IEEE. He is a founding member of the National Software Council and the Washington DC Software Process Improvement Network (SPIN). He is an active speaker on software engineering topics and has served as the Program Chairman and Program Committee member for several conferences. He has numerous publications to his credit. Mr. O'Neill has a Bachelor of Science degree in mathematics from Dickinson College in Carlisle, Pennsylvania.

Contact Information

Don O'Neill
Independent Consultant
9305 Kobe Way
Montgomery Village, Maryland 20886

Phone: (301) 990-0377
email: ONeillDon@aol.com
http://members.aol.com/ONeillDon/index.html

word count: 4,581



NATIONAL SOFTWARE QUALITY EXPERIMENT
A LESSON IN MEASUREMENT
1992-1997

Don O'Neill
Independent Consultant
(301) 990-0377

http://members.aol.com/ONeillDon/index.html

Experiment Purpose

Don O'Neill Consulting

To measure progress towards the national objective

Reduce software problems by a factor of 10 by the year 2000

Set by the DOD Software Technology Strategy in 1992

To benchmark the state of software product quality

Some of the Questions Asked and Answered in the Experiment

To what extent is there a continuing stream of requirements changes?

What are the leading types of errors?

Are errors traced to people or process?

Is a standard development process followed?

To what extent are wrong software functions being developed?

To what extent are there shortfalls in real time performance?

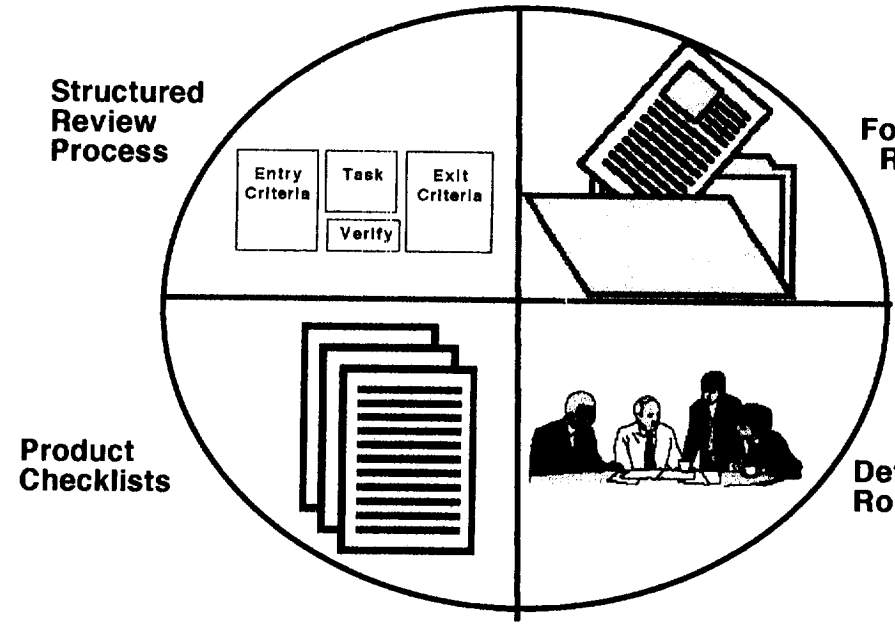
Is gold plating a problem?

Experiment Participants

- Accounting, personnel, administration
- Administrative and management decision support
- Aircraft jet engine diagnostics, logistics, and maintenance
- Artillery fire control system
- Avionics flight on-board control
- Control devices for avionics applications
- Credit card application
- Department of State embassy support
- Electronic commerce
- Electronic warfare
- FAA communications
- Factory line support
- Financial services
- Global positioning system user sets
- Insurance and medical information
- International banking
- Joint Chiefs of Staff support
- Medical information system
- Naval surface weapons system
- Pre and post flight space application
- Telecommunications

Experiment Centerpiece: Inspection Lab

Don O'Neill Consulting



Product Checklist Themes

Don O'Neill Consulting

Completeness
Traceability from code to requirements

Correctness
Intended function with faithful elaboration of steps that carry it out

Style
Naming, commentary, alignment, case, highlighting, templates

Rules of Construction
Application domain specific reference architecture

Multiple Views
Programmer, tester, user, computer resources, security, Y2K

Inspection Reporting Form

Issue Number	Page/ Line	Checklist	Defect Category	Defect Severity	Defect Type	Defect Origin	Defect Description

Defect Category: Missing, Wrong, Extra
Defect Severity: Major, Minor

Defect Type: Interface, Data, Logic, I/O, Performance, Functionality, Human Factors, Standards, Documentation, Syntax, Maintainability, Other

Defect Severity and Category Summary

Don O'Neill Consulting

Defect Severity and Category Summary

	Missing	Wrong	Extra	Total
Major	7.44	5.95	.88	14.27
Minor	49.76	27.63	8.32	85.73
Total	57.20	33.60	9.20	100.00

Inspection Lab Operations Summary

Don O'Neill Consulting

INSPECTION LAB OPERATIONS

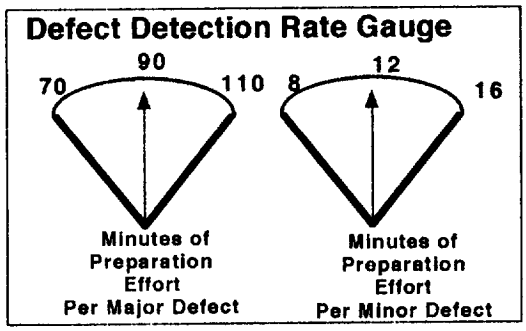
Sessions	Prep Effort	Conduct Time	Major Defects	Minor Defects	Size in Lines
2317	142,306	52,196	1854	9521	788,459

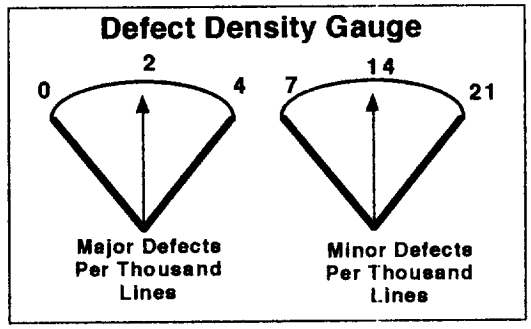
Metrics:

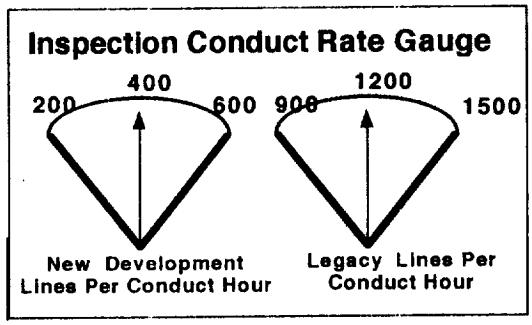
1.	12.51	minutes of preparation effort per defect
2.	76.76	minutes of preparation effort per major defect
3.	2.35	major defects per KSLOC
4.	12.08	minor defects per KSLOC
5.	906	lines per conduct hour
6.	4.91	Defects per session
7.	4.48	Return on Investment

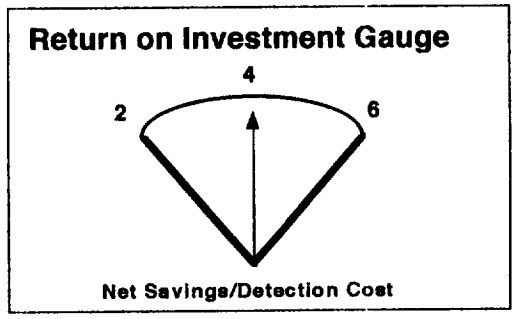
Software Inspections Control Panel

Don O'Neill Consulting



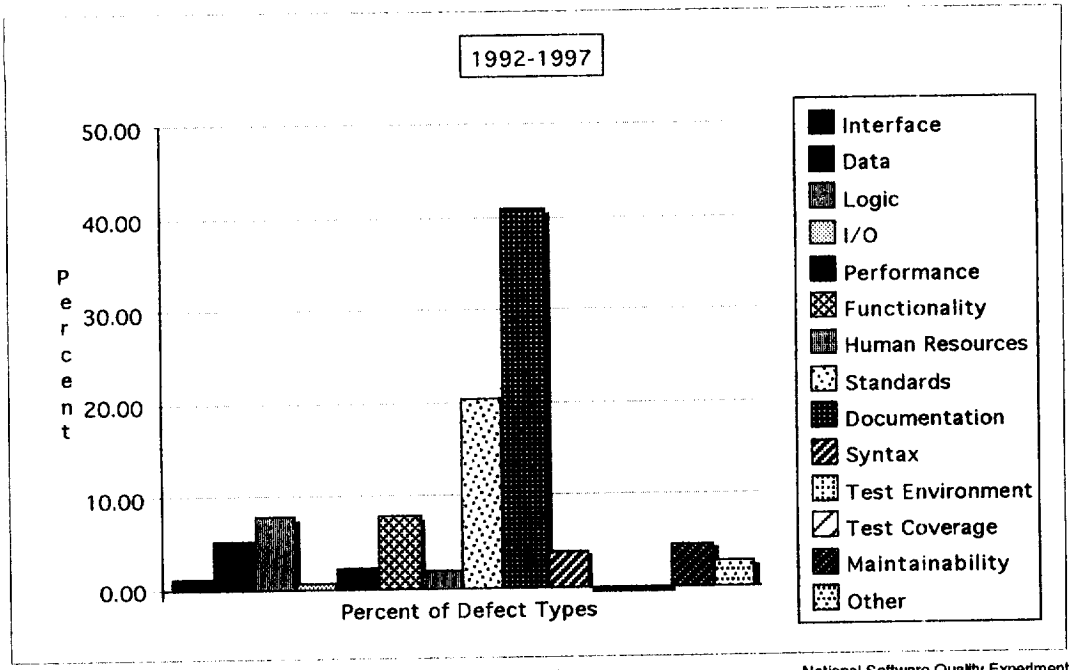






Defect Types

Don O'Neill Consulting



Common Problems

Don O'Neill Consulting

1. Software product source code components are not traced to requirements.

As a result, the software product is not under intellectual control, verification procedures are imprecise, and changes cannot be managed.

2. Software engineering practices for systematic design and structured programming are applied without sufficient rigor and discipline.

As a result, high defect rates are experienced in logic, data, interfaces, and functionality.

3. Software product designs and source code are recorded in an ad hoc style.

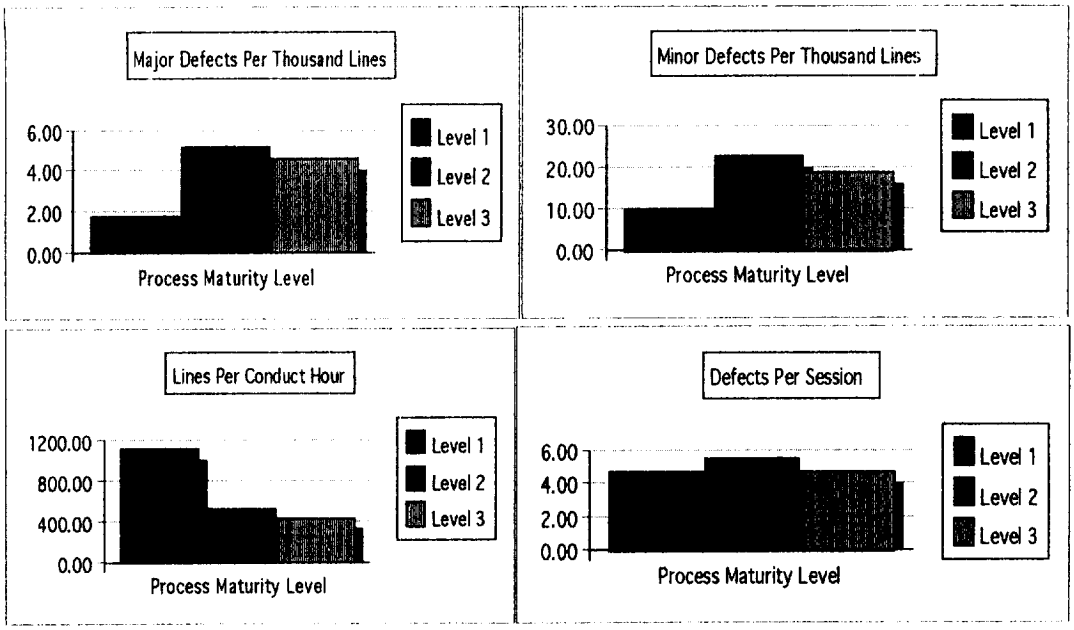
As a result, the understandability, adaptability, and maintainability of the software product are directly impacted.

4. The rules of construction for the application domain are not clearly stated, understood, and applied.

As a result, common patterns and templates are not exploited in preparation for later reuse.

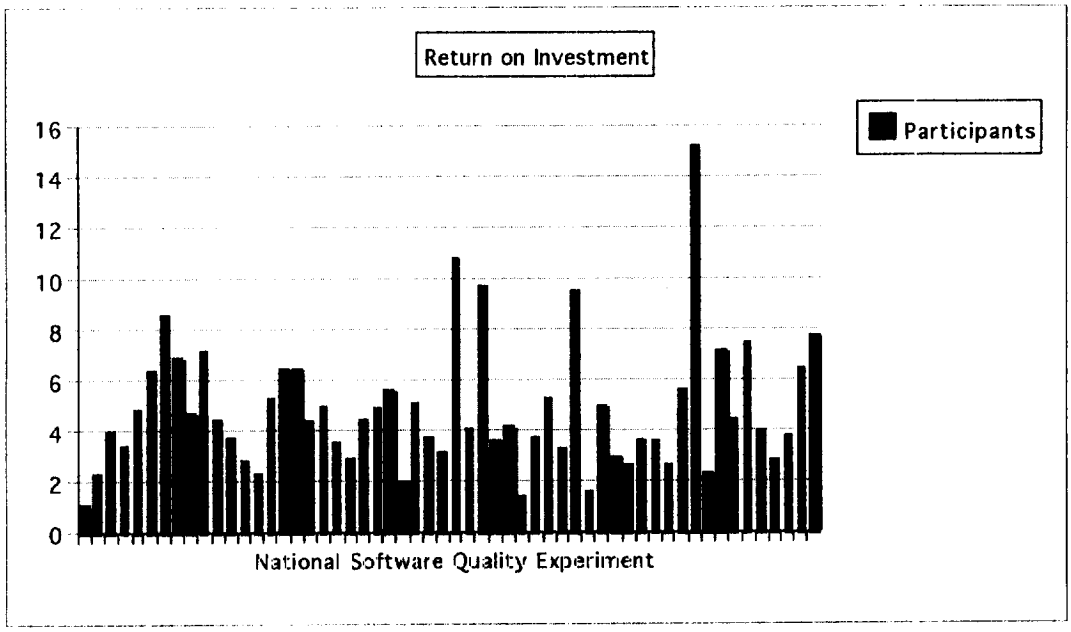
Software Process Maturity Level

Don O'Neill Consulting



Return on Investment

Don O'Neill Consulting



Experiment Findings Summary

Don O'Neill Consulting

Lack of Progress

-The objective to reduce software problems by a factor of 10 is not being met

Looking Harder, Finding More

-By reducing the size of artifacts inspected

Program Size Matters

-Defect density decreases with program size
-Starting, finishing, and fitting in are all more error prone than the body of the program which gives it size

Software Process Maturity Insight

-Legacy software anchors many organizations at level 1
-These are often commercial enterprises

Process Neglect Exceeds Personal Defects

-Organization neglect of its software process exceeds the poor workmanship of individual programmers as the source of errors
-Documentation and standards defect types account for nearly two-thirds of all defects

Return on Investment High

-Software inspections deliver a favorable return on investment with
-Savings exceed costs by 4 to 1

Field Measurement Lessons

Don O'Neill Consulting

1. Measurement must be aligned with business and performance needs.

These activities must be built into the normal operation of the organization.

To do this, the goals to be met and questions to be answered in management, engineering, and operations must precede the collection of data.

2. Metrics must be carefully pinpointed and rigorously defined.

Extraordinary steps must be applied to obtain consistency and uniformity.

Without a well defined process for data collection and analysis, the variance in the measurement process itself impacts the accuracy of results.

3. Attention must be paid to the confidentiality of results.

The opportunity for improvement is increased when the measured results are made more widely available.

-However, individuals and groups naturally resist having their shortcomings made public.
-If ignored, this resistance will defeat the measurement program.
-The organization must strike a balance between public and private data.

NATIONAL SOFTWARE QUALITY EXPERIMENT
A LESSON IN MEASUREMENT

PROLOGUE

The nation's prosperity is dependent on software. The nation's software industry is slipping, and it is slipping behind other countries. The National Software Quality Experiment is riveting attention on software product quality and revealing the patterns of neglect in the nation's software infrastructure.

ABSTRACT

In 1992 the DOD Software Technology Strategy set the objective to reduce software problem rates by a factor of ten by the year 2000. The National Software Quality Experiment is being conducted[1] to benchmark the state of software product quality and to measure progress towards the national objective.

The National Software Quality Experiment is a mechanism for obtaining core samples of software product quality. A micro-level national database of product quality is being populated by a continuous stream of samples from industry, government, and military services. This national database provides the means to benchmark and measure progress towards the national software quality objective and contains data from 1992 through 1997.

The centerpiece of the experiment is the Software Inspection Lab where data collection procedures, product checklists, and participant behaviors are packaged for operational project use. The uniform application of the experiment and the collection of consistent measurements are guaranteed through rigorous training of each participant. Thousands of participants from dozens of organizations are populating the experiment database with thousands of defects of all types along with pertinent information needed to pinpoint their root causes.

To fully understand the findings of the National Software Quality Experiment, the measurements taken in the lab and the derived metrics are organized along several dimensions including year, software process maturity level, organization type, product type, programming language, global region, and industry type. These dimensions provide a framework for populating an interesting set of analysis bins with appropriate core samples of software product quality.

[1] The National Software Quality Experiment is an entrepreneurial activity

Author: Don O'Neill

Don O'Neill is a seasoned software engineering manager and technologist currently serving as an independent consultant. Following his twenty-seven year career with IBM's Federal Systems Division, Mr. O'Neill completed a three year residency at Carnegie Mellon University's Software Engineering Institute (SEI) under IBM's Technical Academic Career Program. There he developed a blueprint for charting software engineering evolution in the organization including the training architecture and change management strategy needed to transition skills into practice.

As an independent consultant, Mr. O'Neill conducts defined programs for managing strategic software improvement. These include implementing an organizational Software Inspections Process, implementing Software Risk Management, and conducting Global Software Competitiveness Assessments. Each of these programs includes the necessary practitioner and management training.

In his IBM career, Mr. O'Neill completed assignments in management, technical performance, and marketing in a broad range of applications including space systems, submarine systems, military command and control systems, communications systems, and management decision support systems. He was awarded IBM's Outstanding Contribution Award three times:

1. Software Development Manager for the Global Positioning Ground Segment (500,000 source lines of code) and a team of 70 software engineers within a $150M fixed price program.
2. Manager of the FSD Software Engineering Department responsible for the origination of division software engineering strategies, the preparation of software management and engineering practices, and the coordination of these practices throughout the division's software practitioners and managers.
3. Manager of Data Processing for the Trident Submarine Command and Control System Engineering and Integration Project responsible for architecture selections and software development planning (1.2M source lines of code).

Mr. O'Neill served on the Executive Board of the IEEE Software Engineering Technical Committee and as a Distinguished Visitor of the IEEE. He is a founding member of the National Software Council and the Washington DC Software Process Improvement Network (SPIN). He is an active speaker on software engineering topics and has served as the Program Chairman and Program Committee member for several conferences. He has numerous publications to his credit. Mr. O'Neill has a Bachelor of Science degree in mathematics from Dickinson College in Carlisle, Pennsylvania.

Principles of Successful Software Inspections

Dennis Beeson and Tim Olson

World-Class Quality
3082 Hamline Ave. N., St. Paul, MN. 55113
Phone: 612-636-2234
Email: DDBeeson@gte.net, Tim.Olson@worldnet.att.net

Abstract

Software inspections remain the most effective method of early defect detection and removal (e.g. early defect detection 80 - 90%, ROI 7:1 - 12:1). Yet many organizations are unsuccessful at invoking the cultural changes required to implement and sustain an effective software inspection process. So what can an organization focus on to change people's perspective of inspections to develop a quality culture centered around software inspections? This paper will identify some of the essential attributes or principles of software inspections which facilitate in building and sustaining a quality culture. This paper will measure the F/A-18 Software Development Team's inspection process against these principles to determine software inspections effectiveness as well as identify areas for future improvement.

Objectives

The objectives of this paper are to:

- present some common cultural problems associated with software inspections.
- present some successful software inspection data from the F/A-18 Aircraft.
- present an overview of effective principles that are successful when performing software inspections.
- benchmark the F/A-18 Software Development Team's inspection process against inspection principles identified to determine effectiveness and indicate areas for improvement.

In This Paper

The following table describes the title and starting page of each section:

The Positive Impact of Inspections on F/A-18

Background

Since 1987, the F/A-18 Software Development Team (SWDT) at the Naval Air Warfare Center - Weapons Division (NAWC-WD) has been providing system and software engineering maintenance and upgrades on the F/A-18 A/B model aircraft Mission Computer (MC) and Stores Management System (SMS) for the US Navy and Foreign Military Sales (FMS) customers.

F/A-18 Mission Computer Upgrades

The F/A-18 SWDT has undertaken four major upgrades to the F/A-18 aircraft's Mission Computer (MC) Operational Flight Program (OFP). The MCs are the center of the F/A-18's avionics architecture. The MCs are the primary link between the aircrews cockpit display environment and the aircraft's tactical and air vehicle management avionics subsystems.

F/A-18 MC Defect Removal Life Cycle

Figure 1 illustrates the overall impact software product inspections and software process improvement have had on product quality. During a ten year period involving over 5000 inspections, early defect detection and defect prevention have significantly moved the defect removal curve to the left. The majority of product defects are now found in the requirements, design and coding phases. In fact, over 86.6% of all defects are found before testing. The defect removal life cycle curve is also used to demonstrate product maturity to the customer.

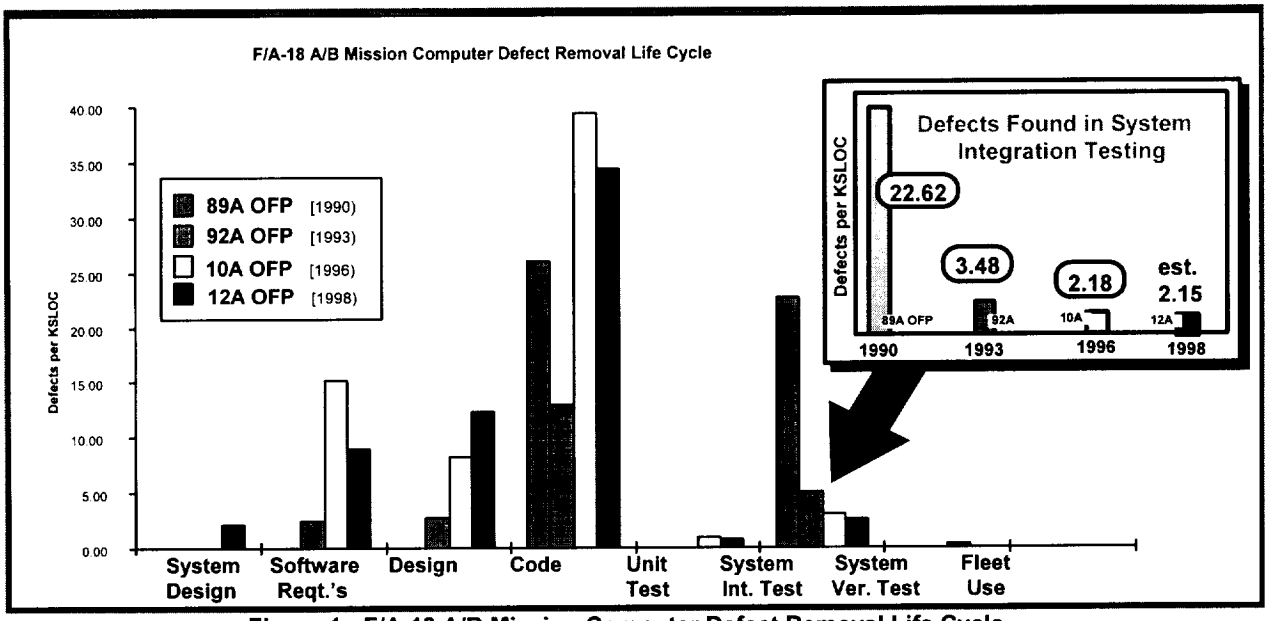


Figure 1: F/A-18 A/B Mission Computer Defect Removal Life Cycle

Benchmarking the F/A-18 Inspection Process

World-Class Software Benchmarks

Benchmarking the F/A-18's inspection process data against a world-class level. Over the last 10 years, the F/A-18 SWDT has progressed from an average performing SEI CMM Level 1 organization to comparing favorably against world-class software organizations. Table 1 characterizes current performance of various world-class organizations to the F/A-18 SWDT current performance capability.

Measurement	World-Class Benchmark*	F/A-18 Software Development Team
Quality		
Inspection Defect Removal Efficiency	80%-90%	86.6%
Post-Release Defect Rate	.01 per KSLOC	.01 per KSLOC
Cost		
Total Cost Savings	$7.5-$45 Million	$14.4 Million ($ 3.6M per major update)
Inspection Cost	$2,500 on Average	$1,500 on average
Return on Investment (ROI)	7:1 - 12:1	7:1
Schedule		
Schedule / Cycle Time	Reduced 10-25% per yr.	Reduced 9% per year
Productivity	Doubled in 3 years	Increased 62% in 3 years

Table 1 World-Class Software Benchmarks *derived from World-Class Quality - Timothy G. Olson copyright 1995 - 1996

Principles of Successful Software Inspections

Principles of Software Inspections

To fully understand how to optimize software inspections to promote team building and improve individual learning it was necessary to have a clear description of the core attributes or principles that make software inspections successful from a people prospective. Only after these principles were identified was it possible to make the necessary process improvements. Research and benchmarking of software inspections best practices were successful in identifying the following principles found in most effective inspection processes:

Principles	Description
Leadership	Management should provide resources and be an active participant in communicating, mentoring, and building the organizations quality culture. Facilitate the team in setting clearly stating mission, goals, and objectives centered around quality, quality measurement, and quality improvement.
Quality Culture	Foster commitment to designing in quality. Develop an understanding of the quality expectations, values, and priorities of the immediate and final customers.
Responsibility	Foster responsibility for the quality of the end product
Process Ownership	Team participation in process definition and process change mechanisms.
Defect Prevention	Foster commitment to learning from past defects.
Communication	Foster open honest communication supported by effective meeting facilitation. Understand the strength and weaknesses of self, team, and organization and use this diversity to optimize effectiveness. Operate organization with integrity, making decisions based on what is truly best for product quality and the organization.
Feedback	Give feedback on individual defects found, overall product quality, status of defect prevention (e.g. common defect trends identified, changes to data driven checklists).
Defect Analysis	Analysis and tracking of defect density per development phase and determining criteria for reinspection.
Agreement	Management, engineering, suppliers, and immediate and final customers should effectively review and agree to product plans (e.g. schedule, resources, staffing, quality objectives, etc..).
Defined Process	Fully communicate what is expected of management, engineering, suppliers, and immediate and final customers (e.g. what, how, when, were, why).
Training	Effectively train people in inspection purpose, roles, process, facilitating meetings.
Defect Identification	Formal mechanism for documenting, categorizing, and dispositioning defects. Defect identification involves gathering defect and associated metrics (e.g. size, effort, cost, time, rework). Defect identification is usually supported by data driven checklists.
Accountability	Formal mechanism hold developers, reviews, and moderators accountable for fulfilling their role in the inspection process.

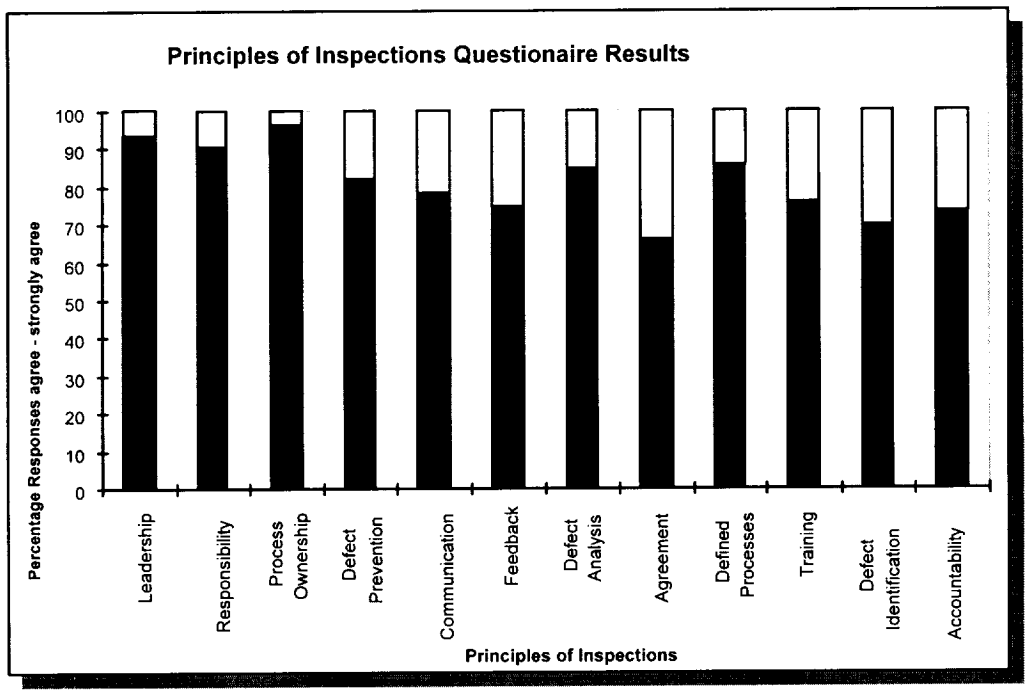
Measuring the Principles of the F/A-18 Inspections

F/A-18 Software Team

Over the last ten years the F/A-18 Software Development Team has training approximately 50 software engineers in formal inspections. Most have never used formal inspection methods before working on the team. As they progress in knowledge and understanding of inspections they move up in their level of commitment to the teams product quality goals and buy-in to the inspection process. The principles of software inspects need to be effective and in place to protect against loosing buy-in or commitment, issues of non-compliance, or to assist in gaining enough trust in the team and the inspection process to move to a higher level of buy-in or commitment.

Questionnaire

A survey was conducted of the F/A-18 Software Development Team in order to measure the buy-in and commitment to the software inspection principles. The table below shows the results:



Summary

Achieving measurable results using software inspections requires understanding fundamental principles, and then tailoring those principles to practice. These principles must then become part of an organization's day to day business.

References

References

The references used for this presentation are:

- [Covey 91] S. R. Covey, Principle Centered Leadership, New York, NY: Simon & Schuster, 1991
- [Senge 90] P. M. Senge, The Fifth Discipline, New York, NY: Currency Doubleday, 1990
- [Curtis 95] Curtis, Bill, et al. People *Capability Maturity Model* (CMU/SEI-95-MM-02). Pittsburgh, PA: Carnegie Mellon
- [Deming 95] W. E. Deming, Out of Crisis, Cambridge, MA: MIT Center for Advanced Engineering Study, 1995
- [Beeson 98] D. D. Beeson and T. G. Olsen, "Benchmarking F/A-18 Software Inspection Data", SEI 1998 Conference Proceedings, 1998.
- [Barnard 94] Barnard, J. and Price, A. "Managing Code Inspection Information", IEEE Software, March 1994.
  [Billings 94] Billings, C., et al. "Journey to a Mature Software Process", IBM Systems Journal, vol. 33, no. 1, 1994.
- [Ebenau 94] Ebenau, B. and Strauss, S., *Software Inspection Process.* McGraw-Hill, 1994.
- [F/A-18 96] F/A-18 MC/SMS Software Processes; February 12, 1996.
- [F/A-18 97] F/A-18 Systems Engineering Process Guide; August 9, 1997.
- [Fagan 76] Fagan, M. "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems J., no. 3, 1976. pp 182-210.
- [Fagan 86] Fagan, M. "Advances in Software Inspections", IEEE Transactions on Software Engineering, July 1986
- [Gilb 93] Gilb, T. and Graham, D. *Software Inspection.* Addison-Wesley, 1993.
- [Grady 94] Grady, R. and Van Slack, T. "Key Lessons In Achieving Widespread Inspection Use", IEEE Software, July 1994.
  [Herbsleb 94] Herbsleb, James, et al. "Benefits of CMM-Based Software Process Improvement: Initial Results", CMU/SEI-94-TR-13, 1994.
- [Humphrey 89] Humphrey, W. S. *Managing the Software Process.* Reading, MA: Addison-Wesley Publishing Company, 1989.
- [Olson 94] Olson, Timothy G., et al. "A Software Process Framework for the CMU/SEI-94-HB-01, 1994.
- [Olson 96] Olson Timothy G., "World-Class Software Inspections", SEI 1996 SEPG Conference Proceedings, 1996.
- [Paulk 93] Paulk, Mark C., et al. *Capability Maturity Model for Software, Version 1.1* (CMU/SEI-93-TR-24). Pittsburgh, PA: Carnegie Mellon University, 1993.
- [O' Hara 97] F. O'Hara, Achieving maximum benefits from formal reviews/inspections - strategies and case studies, proceedings of EuroSTAR'97, Edinburgh and SPI'97, Barcelona, 1997.

Principles of Successful Software Inspections

NASA/Goddard Software Engineering Workshop

Presented by

Dennis Beeson
F/A-18 Software Development Team, Manager
Ki Solutions Consulting, Co-Founder
SEI Certified SCE Evaluator
(760) 375-3376
DDBeeson@gte.net

Tim Olson, President
World-Class Quality
Juran Institute Associate
Authorized SEI Lead Assessor
(612) 636-2234
Tim.Olson@worldnet.att.net

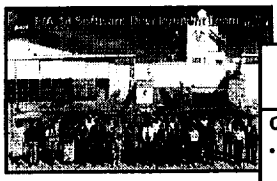
Objective

- Provide principles of effective software inspections derived from real-world organizations
- Example using inspection principles to benchmark:
  - Inspection process
  - Individual buy-in and commitment

Agenda

- F/A-18 Software Team Overview
- Software Inspection Principles Identified
- Benchmarking Inspection Process
- Inspection Principles and Buy-in
- Measuring People Buy-in and Commitment
- Question ????

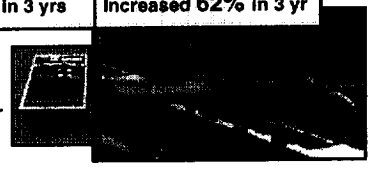
F/A-18 Software Team Overview



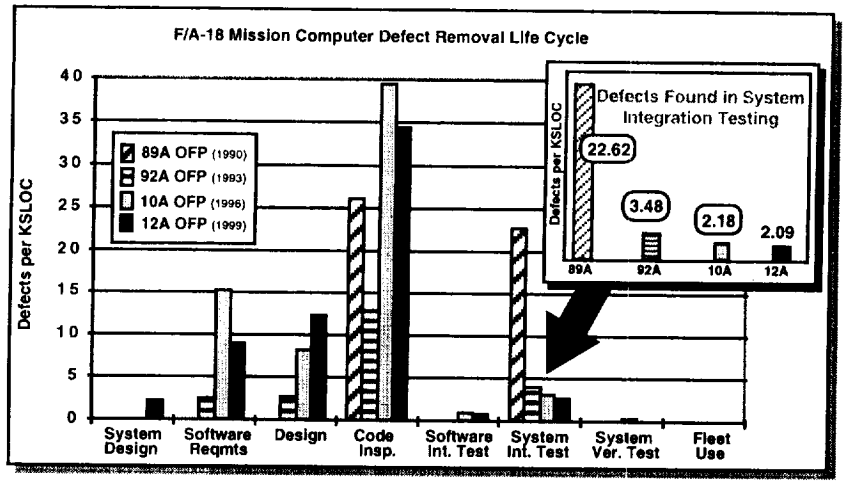
SEI CMM Level 3 rating

Measurement	World-Class Benchmark	F/A-18 Software Development Team
QUALITY		
• Inspection Defect Removal Efficiency	80% - 90%	86.6%
• Post-Release Defect Rate	.01 per KSLOC	.01 per KSLOC
COST		
• Total Cost Savings	$ 7.5M - $ 45M	$14.4 Million
• Inspection Cost	$ 2500 on Avg.	$1500 on Avg.
• Return on Investment (ROI)	7:1 - 12:1	7:1
SCHEDULE		
• Schedule / Cycle Time	Reduced 10-25% yr	Reduced 9% per yr
• Productivity	Doubled in 3 yrs	Increased 62% in 3 yr

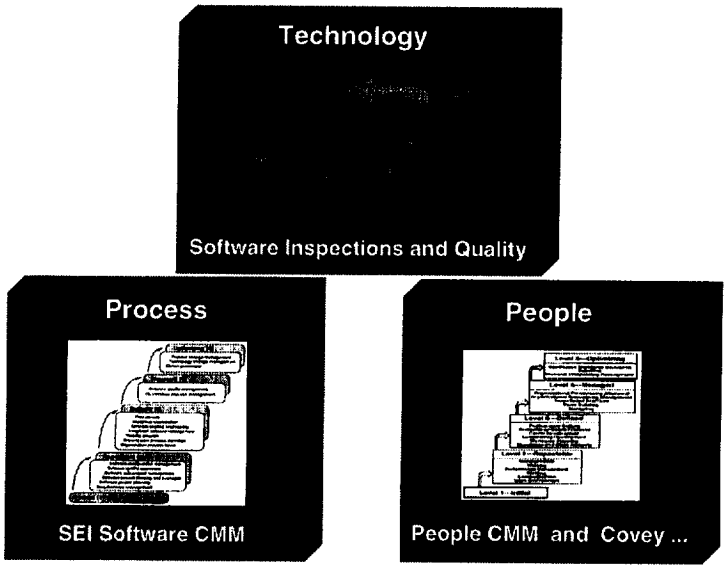
Mission Computer



F/A-18 Software Team Performance



Focusing on software inspections has reduced product defects by 86.6%



Principles of inspections involves: Technology, People, and Process

Principles of Software Inspections Identified

Leadership	foster, communicate, mentor, and facilitate a quality culture
Responsibility	personally identifies with quality of product
Process Ownership	willing to take on process improvement
Defect Prevention	root cause analysis of common defects for data driven checklists
Communication	facilitated meetings, environment focused on product quality
Feedback	author defects, product defect density
Defect Analysis	defect density per development phase and reinspection criteria
Agreement	agreement to plans and tasking
Defined Process	clear description of what to do when
Training	re-enforcement of what to do when and why
Defect Identification	document, categorize, and disposition defects
Accountability	reinspection criteria and moderator tracking defects to closure

Benchmarking the Inspection Process

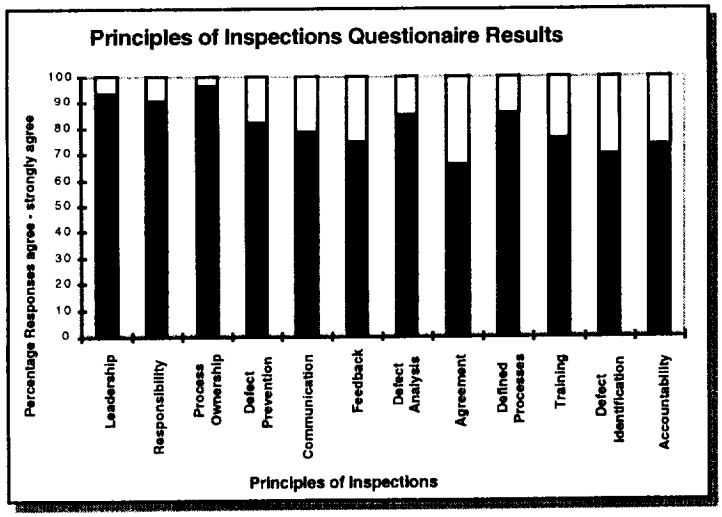
Leadership Responsibility Process Ownership Defect Prevention	• **Quality Definition** (e.g. Conformance to customer requirements, meeting or beating defect removal lifecycle removal curve) • **Define, document, and train defect prevention process**
Communication Feedback Defect Analysis	• Add overview meeting to educate reviewers on inspection package • **Insure feedback on project defect density per phase** • **Reinspection criteria** (e.g. 10 major defects found or low preparation rate)
Agreement Defined Process Training Defect Identification	• **Review development plan with software engineers** • Add moderator training stressing facilitation skills and inspection principles • Update general inspection training class with inspection principles • Data driven checklists to educate reviewer on common defects
Accountability	• **Preparation rate set a 10 - 15 pages per hour** • **Entry criteria for review material** (e.g. checklist of items, spell checked, clean compile)

Software Principles and Buy-in

Commitment	Does whatever is needed including creating new norms	
Enrollment	Does whatever can be done within the norms	
Genuine Compliance	Sees benefit, does what is expected and more	Leadership Responsibility Process Ownership Defect Prevention
Formal Compliance	Sees benefit, does what they are told	Communication Feedback Defect Analysis
Grudging Compliance	Does not see benefit, does not want to lose their job	Agreement Defined Process Training Defect Identification
Non-Compliance	Won't do it	Accountability
Apathy	Does nothing	

Commitment based on effective principles

Measuring Peoples Buy-in and Commitment



Questions ????

Capture-Recapture – Models, Methods, and the Reality

Jens-Peder Ekros[1] jenek@ikp.liu.se
Anders Subotic[2] andsu@ida.liu.se
Bo Bergman[1] bober@ikp.liu.se

[1]Division of Quality Technology and Management

[2]Department of Computer and Information Science
Applied Software Engineering Laboratory

[1,2]Linköping University,
SE-581 83 Linköping, Sweden

59-61

Abstract

Software inspections are widely used for defect detection, and are capable of detecting defects early in development. In order to avoid spending too much resource and to assure that the inspected product has the demanded quality, a method to estimate product quality and inspection performance would be helpful. For this purpose, capture-recapture methods have been suggested. In this paper, we explore the relation between models underlying capture-recapture methods and inspection data. We have tested three hypotheses that underlie commonly used capture-recapture methods: Inspectors find the same number of defects; Defects are equally easy to detect; and, Inspectors find the same defects. We find no support for any of the three hypotheses. The paper also contributes to research by describing methods for testing the hypotheses. It is not wise to generalise from these results, as the sample analysed is small. Nevertheless, the results imply that the underlying models, or assumptions, of commonly used capture-recapture methods are not generally applicable to software engineering.

Introduction

Software plays an important role in today's society. The high dependence on software has put focus on software quality engineering. Customer satisfaction through good quality gives competitive advantage. Further, lack of quality costs, especially if quality deficiencies remain undetected from early phases of development. The lowest level of quality engineering is the detection of defects for the sole purpose of correction. The second level is quality assurance, where product measurements is compared to standards so as to assure that the shipped product is of the "right" quality. To assure is more demanding than to detect, and requires models of product quality.

Software inspection is a family of widely used methods for defect detection, capable of detecting defects early in development. In many organisations that develop software, inspections are an essential part of the process. It has been recognised that inspections have a positive effect on product quality as well as the efficiency of the development process. However, inspections demand time and resources. Preparations must be made before the inspection meeting where many key persons will attend. This is a problem. In order to avoid spending too much resource and to assure that the inspected product has the demanded quality, a method to estimate the performance of the inspection would be helpful.

Several approaches based upon different statistical techniques have been evaluated in order to get better basis to assess the above mentioned aspects. Briand et al. (1997) described three approaches:

1. Comparing inspection results with historical defect count.
2. Comparing inspection results with a baseline for defect density.
3. Estimating the number of residual defects using the current inspection results.

This paper addresses issues related to the third approach.

Lately, capture-recapture methods have gained increased attention in the software engineering community, see e.g. Eick et al. (1992). The purpose of capture-recapture methods is to estimate the size of populations.

These methods have their origin in the biological research society, recent examples include Chao (1988), Chao et al. (1992), and Pollock (1991). The methods have also been used outside of the biology area. For example, Efron and Thisted (1976) estimated the number of words known by Shakespeare.
Adapting methods to new areas, i.e. software inspections, may lead to difficulties which one has to have in mind. The most important aspect of adaptation is that of the underlying models. The different estimation methods assume certain conditions on the data, i.e. specific models. If the model does not correspond to the data, the method may give results that are either incorrect or easily misinterpreted.

From related work we have found that there sometimes is a lack of distinction between models and methods, also known as estimators, or the issue is not mentioned altogether. The bulk of work on capture-recapture in the field of software engineering has been concerned with methods, e.g. Vander Wiel and Votta (1993), Wohlin et al. (1995), Briand et al. (1997), Miller 1998, and Wohlin and Runesson (1998). Assumptions have been made, and sometimes claimed, with little support from literature or analyses, e.g. Vander Wiel and Votta (1993) and Wohlin et al. (1995). There are few tests of consequences of broken model assumptions on results, e.g. Vander Wiel and Votta (1993). However, analyses investigating how the assumptions behind methods correspond to reality are missing.

In this paper we examine published inspection data sets in order to learn more about inspector capability, defect detectability, and how these relate to each other. The importance of this work is that it provides a basis for use of prediction methods, by validating, or invalidating, model assumptions demanded by methods. In order to manage this, new variants of statistical tools have been used. The differences between inspection data and that of other fields, e.g. a low density of information in the tables, increases the difficulty of conducting tests on inspection data. One problem is that ordinary distributions cannot be used. This forces the creation of specific distributions for each specific case. These distributions depend on the size of the table, the number of rows and columns, as well as the density of the table, i.e. the ratio of ones. A number of extended computer simulations of Monte Carlo type and enumeration were conducted in order to create these distributions.

Background

Generally, capture-recapture based estimation of population size begins with sampling of the population. The results of sampling are used as parameters in an estimator function, which gives the size of the population, if certain conditions are fulfilled. In previous published work in the field of software engineering, the main focus was investigation of the performance of different methods, or estimators. In fact, methods and models are often confused. In this paper, an estimator, or method, denotes the way in which an estimate is computed. A model represents a set of assumptions on input data under which a method has been designed to work.

The most common families of models are:

1. $p_{ij} = p$: the probability of an inspector having detected a defect is constant and does not vary with inspector or defect.
2. $p_{ij} = p_i$: the probability depends on the difficulty of the defect, which varies between defects, and all inspectors have the same capability of detecting a specific defect.
3. $p_{ij} = p_j$: the probability depends on the capability of the inspector, which varies between inspectors, and all defects are equally difficult to detect for a given inspector.
4. $p_{ij} = p_i p_j$: the probability depends on the capability of the inspector as well as the difficulty of the defect, which both vary.
5. $p_{ij} = p_{ij}$: the detection probability might be individual depending on both inspector and defect.

Miller (1998) gave additional assumptions that relate to the process of (re-)capturing.

The above mentioned models are implicitly used in estimators. The most common estimators are Jack-knife, Maximum-likelihood, and the Chao estimator, of which there are several versions. The Jack-knife estimator is based on model number two, Maximum-likelihood on number three, and Chao estimators exist for numbers two to four.

Vander Wiel and Votta (1993) studied "the effects of broken [model] assumptions on" the Jack-knife and Maximum-likelihood estimators. The Maximum-likelihood estimator was found to perform better than Jack-knife, especially if defects were grouped to achieve homogenous detectability. Wohlin et al. (1995) claimed that the assumptions of the Jack-knife method do not correspond to reality and rejected it in favour of the Maximum-likelihood method. The claim was not supported by a test. They also evaluated a filtering technique to improve estimates of residual defects. The new method was evaluated using data from an experiment where a single document was inspected. Briand et al. (1997) examined the sensitivity of methods with respect to the number of samples used, i.e. number of inspectors. They recommended that at least four inspectors be used, and that the Jack-knife estimator was the best for four or five inspectors. The Chao estimator with the same model as Jack-knife was comparable for five inspectors, but behaved badly for four inspectors. The evaluation criticised by Miller (1998) with respect to choice of inspectors and number of data points. Wohlin and Runesson (1998) proposed two new estimation methods based on extrapolation of fitted curves. The methods are based on a number of assumptions that are not tested. The methods were evaluated with inspection data from two experiments, where the choice of artefacts was criticised by Miller (1998). Miller (1998) arrived roughly at the same conclusions as Briand et al. (1997). However, Miller recommended Jack-knife for three to five inspectors and for six inspectors both Jack-knife and the Chao estimator for model number four. In conclusion, there are no known examples in software engineering literature where the viability of the models assumed by capture-recapture methods is tested.

Models and Tests

By using capture-recapture methods on inspection data we want to predict, or assess, the remaining number of defects in the inspected document, the performance of the inspection, or both. To facilitate analysis of empirical data from a certain perspective, models that faithfully represent the data are needed. These models are the basis for analysis methods. That is, the " mathematical model ... relates the attributes to be predicted to some other attributes that we can measure now. " (Fenton and Pfleeger 1996)

A number of model assumptions have implicitly been made when a method has been chosen. The methods are dependent on the underlying models that supposedly describe the data to be analysed. This is important but often forgotten. In this section we will present ways to determine model characteristics of inspection data. This gives a better basis for choosing or creating suitable estimation methods. By looking at published results from a number of inspections some conclusions regarding the underlying models have been made. We have also made a contribution in the methods to determine these models.

The type of defect-inspector table used throughout this paper is shown in Figure 1. The table represents the defects found as r rows and inspectors that found the defects as c columns. Let n_{ij} be the contents of a cell representing the detection of defect i by inspector j. If the defect on row i was detected by inspector j, n_{ij} is one, otherwise n_{ij} is zero. The number of inspectors that detected defect i is the number of ones in row i, the row sum, denoted $n_{i.}$. The number of defects detected by inspector j is the number of ones in column j, the column sum, denoted $n_{.j}$. The total number of detections is denoted $n_{..}$.

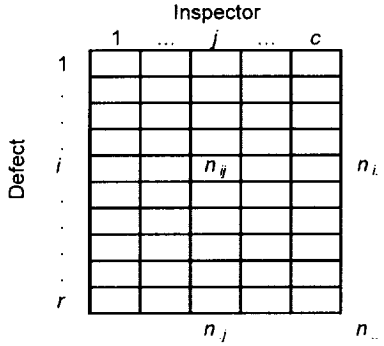


Figure 1. Graphical description of an inspection data matrix.

In the rest of this section we describe tests for analysing inspection data, mainly with respect to aspects that are relevant to the most commonly used capture-recapture methods. Interesting aspects of inspection data relate to inspectors, defects, and their relation.

Inspector Capability and Defect Detectability

The assumptions regarding the capabilities of inspectors are concerned with the number of defects detected by each inspector. Intuition often suggests that inspectors have different capabilities but this has not yet been tested, see e.g. Wohlin et al. (1995). The capability of inspectors is tested by comparing the number of defects found by each inspector with the average number of defects detected.

A test statistic similar to the Chi-square test (Everitt 1992) is utilised, $Q = \sum_{j=1}^{c} \left(n_{.j} - \frac{n_{..}}{c} \right)^2 \Big/ \frac{n_{..}}{c}$. Problems arise when the expected values of row or column sums are too small. The Chi-square test is commonly considered to work less than well for values below five. With respect to row sums, this criterion is not fulfilled by any of the tables analysed in this paper. The expected values of column sums fulfil the criterion for roughly half of the tables. Preferably, under the null hypothesis of equality, the statistic for an observed table is compared with a reference distribution. Since the Chi-square distribution is not applicable, a substitute distribution has to be constructed.

Enumeration was the approach chosen for creating the distribution of the Q-statistic. That is, in e.g. analysing inspector capability, the set of column sums of a table is analysed. The total number of detected defects, the number of 1:s in a table, are distributed in all unique ways over the columns. The Q-statistic for each permutation is computed and its occurrence is weighted by its probability under the assumption of homogeneity. The result is a reference distribution adapted to characteristics of the table. The p-value for the Q-statistic of the observed table, Q_o, is obtained from the distribution as $p = P(Q > Q_o)$, where the variable Q belongs to the distribution.

Similarly to inspector capability, defect detectability is defined as the fraction of inspectors that found a specific defect. That is, a defect found by many inspectors is said to have a higher degree of detectability. Wohlin et al. (1995) recommended that defects be divided into two groups based on the number of inspectors that found each defect. When only one inspector found a defect the defect was put in a low detectability group. Defects found by more than one inspector were put in a high detectability group. However, this reasoning hides the assumption that defects have different probability of detection. The test for difference among defects is the same as for inspector but along the other dimension of the table.

Defects and Inspectors Combined

A third approach to test inspection data is to consider defect and inspector characteristics at the same time. That is, the test helps determine if inspectors are equally good in finding different types of defects. Since the cell values of the tables analysed are either one or zero, chi-square tests are not appropriate. An alternative test statistic is proposed. The new test statistic is built up by a sum of the differences between the inspectors based on every specific defect.

Let l and m be the identities of two inspectors. Let n_{ij} be one if defect i was detected by inspector j, and zero otherwise. A matrix K is created where $k_{lm} = \{number\ of\ n_{il} > n_{im}\}$ where $l \neq m$ and $k_{ll} = \{number\ of\ n_{il} > 0\}$. A proposed test statistic is $T = \sum_{l} \sum_{m} \left(k_{lm} / k_{ll} \right) * w_l$ where $w_l = 1$.

The diagonal of K, k_{ll}, represents the number of defects found by each inspector. Values outside the diagonal i.e. k_{ij} represents the number of defects found by inspector i but not by inspector j. That is, a big number outside the diagonal indicates that one of the inspectors found many defects not found by the other. The matrix is quadratic, but it is rarely symmetrical, as e.g. inspectors seldom find the same number of defects.

The distribution of the T-statistic is not known, but needed in order to determine the meaning of a T-value. Thus, the distributions of T for each table have to be generated. Due to problem size, enumeration is not an option. Instead, Monte Carlo simulation is used. That is, the distribution of T for a given table is acquired by computing the T-statistic for n randomly generated tables, with similar characteristics as the observed

table. The simulations are crucial to the reliability of the analysis, and so the bulk of work has been spent on trying to achieve a simulation that represents the true probability distribution. Again, the rationale for simulations is that the distribution of the test statistic is unknown.

Results

In this section we investigate characteristics of published inspection data sets and in the process we provide ways of testing model assumptions. The data sets were mainly taken from Freimut (1997), where the majority originates from experiments using NASA subjects. A data set from Wohlin et al. (1995) and one from Myers (1978) were also used in some of the analyses. The results are used for accepting or rejecting the hypotheses stated above. Three different hypotheses are tested.

Inspector homogeneity

For each table *k* the test-variable Q_k is calculated. The value Q_k is compared to an empirical distribution constructed using enumeration, as described above. For this analysis, two data sets were not used due to time and size complexity problems, as there were many combinations to enumerate.

Under the null hypothesis of homogeneity, the expected value of p is 0.5. A low p-value is the result of large differences between the number of defects discovered by different inspectors. A single small p-value supports rejection of the null hypothesis but is not enough to safely reject it. By combining analyses of a number of tables we get a greater body of evidence. Figure 2 shows the p-values for the 22 different data tables tested. Under the null hypothesis the p-values would be evenly distributed between 0 and 1. This does not seem to be the case. The conclusion is therefore that the null hypothesis is rejected. Thus, based on this set of data, we can say that inspectors generally do not find the same number of defects, i.e. inspector capability varies with inspectors.

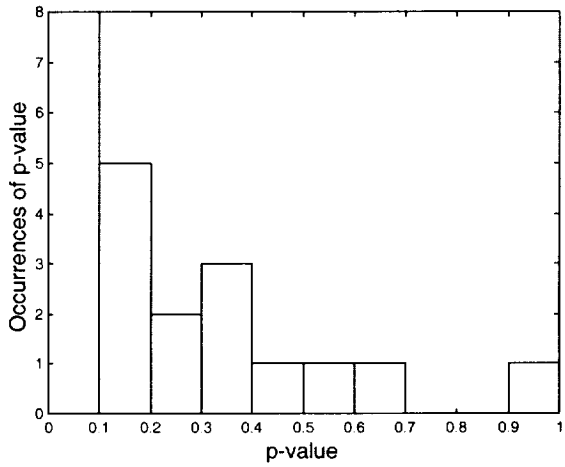


Figure 2. p-values for inspector homogeneity.

Defect homogeneity

Analysing the defect detectability in the same way as inspector capability shows that it can be concluded that defects do not have equal detectability. The p-values for the 22 data sets shown in Figure 3 clearly indicate a non-equal distribution. That is it cannot be said that the different defects generally have equal detectability. For this analysis, two data sets were not used due to time and size complexity problems, as there were many combinations to enumerate.

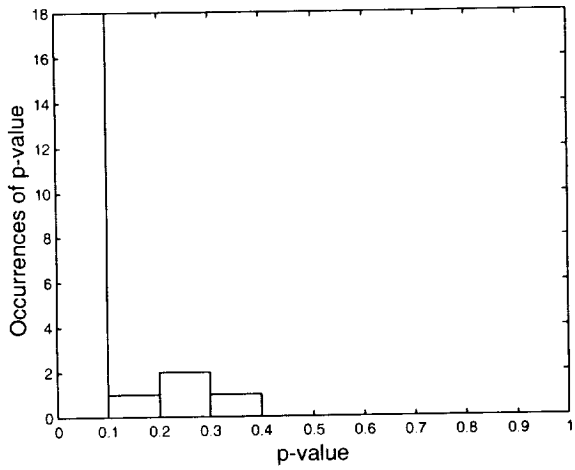


Figure 3. p-values for defect homogeneity.

Another way of representing the p-values is with a scatter plot, shown in Figure 4. Here p-values for inspector capability and defect detectability for 22 tables are plotted against each other. There are no real outliers. That is, no table plots in the upper right quarter. The plot gives stronger support for rejection of the null hypothesis for defect detectability than for inspector capability. That is, defects are more heterogeneous than inspectors. Even though inspectors seem to be more homogeneous than defects, they are predominantly heterogeneous.

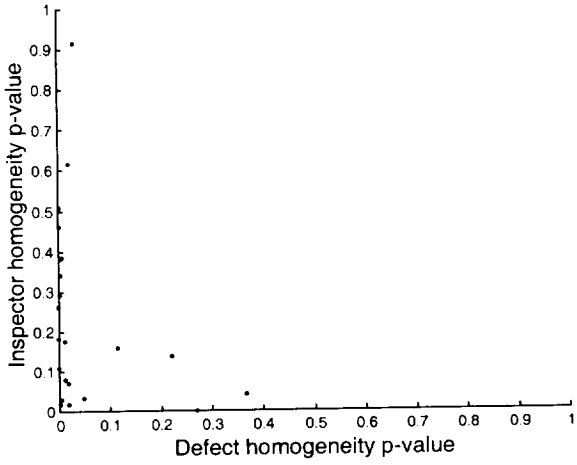


Figure 4. Scatter plot of defect and inspector homogeneity p-values.

Inspector and defect combined homogeneity

In this section, we analyse the relations between defects and inspectors. In Figure 5 the p-values from this analysis is presented. The results stem from 10.000 simulations of 24 tables. As in the other cases it can clearly be seen that the p-values are not evenly distributed between 0 and 1. A low p-value is the result of large differences between inspectors; i.e. inspectors find different defects. The distribution of p-values is skewed towards zero. In fact, 50 percent of the values are lower than 0.1 and about 85 percent are less than 0.5. This situation is highly unlikely under the assumption that inspectors find the same defects. The implication is that there is no support for "inspector profiles", i.e. groups of inspectors that find similar subsets of the defect population. If groups of inspectors found largely the same defects p-values should be skewed toward the right. It may be harsh to reject the concept of "inspector profile" based solely on this analysis. However, analytical advocacy is no longer enough.

This is analysis differs from the test of inspector homogeneity above, where it was shown that different inspectors find different number of defects.

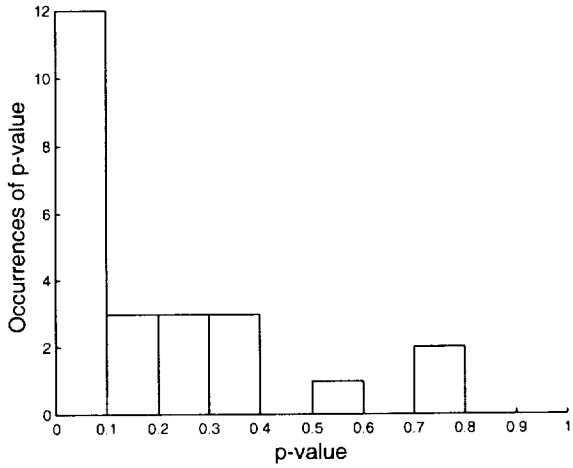


Figure 5. The p-values for combined inspector and defect homogeneity.

Conclusion and Discussion

In this paper we went back to the basics, that is, using the information from inspections to explore the underlying models that are assumed to characterise inspection data. The road towards better methods and correct use of existing methods will start from the most logical point, the distribution and properties of the data.

We have tested three hypotheses occurring in capture-recapture work in the software engineering field:

1. Inspectors find the same number of defects.
2. Defects are equally easy to detect.
3. Inspectors find the same defects.

We find no support for any of the three hypotheses. These results imply that the underlying models, or assumptions, of commonly used capture-recapture methods are not applicable to the software engineering data analysed in this paper. Even though 24 data sets were analysed, 16 of these originate from NASA. Thus, it is not wise to generalise from these results. However, the results suggest that it is wise to test model assumptions. Testing model assumptions requires good data, which in turn requires good instrumentation and collection procedures. We have instrumented the inspection process of an industry partner, and are currently awaiting data to accumulate.

By exploring data from several inspections we are able to draw general conclusions about how the data is formed. This includes measures of correlation between inspectors and between faults. Other aspects are the distributions of inspectors' performance and defect detectability. Naturally, it may not be possible to find a single specific model that will explain all relationships between inspectors and defects. The data sets used here, to estimate characteristics of product and inspection process, have only two parameters. It is not unlikely that other product, process and resource attributes could increase the usefulness of estimators. There may be important differences between instances of inspections, e.g. differing inspection rate, team expertise, type of document, and organisational culture.

Even though universal models are few and far between, the goal is to find general models. It may be easier to find methods to derive situation specific models. These methods can then be used together with local inspection data to assure that a suitable estimation method is used. Still, it is questionable how the information gained can be used. Does the estimate of defect content depend mainly on the product, the measurement process, or both? If the inspection process is unstable, measurement noise may obscure

product attributes. For example, it is hard to determine if a high defect count is the result of a bad product, a good inspection, or both. An accompanying metric is needed for normalisation.

Acknowledgements

The authors wish to thank Mary Helander for her helpful comments on this paper. This work was supported by the Swedish National Board for Industrial and Technical Development (NUTEK), administrated by the Swedish Institute for Applied Mathematics, and the Swedish Foundation for Strategic Research through the ECSEL graduate school at Linköping University, Sweden.

References

Briand, L. C., Emam, K. E., Freimut, B., and Laitenberger, O. (1997). "Quantitative Evaluation of Capture-Recapture Models to Control Software Inspections." *Report 97-22*, ISERN.

Chao, A. (1988). "Estimating Animal Abundance with Capture Frequency Data." *Journal of Wildlife Management*, 52(2), 295-300.

Chao, A., Lee, S.-M., and Jeng, S.-L. (1992). "Estimating Population Size for Capture-Recapture Data When Capture Probabilities Vary by Time and Individual Animal." *Biometrics*, 1992(March), 201-216.

Efron, B., and Thisted, R. (1976). "Estimating the number of unseen species: How many words did *Biometrika*, 63(3), 435-447.

Eick, S. G., Loader, C. R., Long, M. D., Votta, L. G., and Vander Wiel, S. (1992). "Estimating Software *Proceedings of the Fourteenth International Conference of Software Engineering*, May, Melbourne.

Everitt, B. S. (1992). *The Analysis of Contingency Tables*, Chapman & Hall, London.

Fenton, N. E., and Pfleeger, S. L. (1996). *Software Metrics: A Rigorous and Practical Approach*, International Thomson Computer Press, London.

Freimut, B. (1997). "Capture-Recapture Models to Estimate Software Fault Content," *Masters Thesis*, University of Kaiserslauten.

Miller, J. (1998). "Estimating the number of remaining defects after inspection." *Report 98-24*, ISERN.

Myers, G. J. (1978). "A Controlled Experiment in Program Testing And Code Walkthroughs/Inspections." *Communications of ACM*, 21(9), 760-768.

Pollock, K. H. (1991). "Modeling Capture, Recapture and Removal Statistics for Estimation of Demographic Parameters for Fish and Wildlife Populations: Past, Present, and Future." *Journal of the American Statistical Association*, 86(413), 225-238.

Vander Wiel, S. A., and Votta, L. G. (1993). "Assessing Software designs using Capture-Recapture Methods." .

Wohlin, C., and Runesson, P. (1998). "Defect Content Estimations from Review Data." *Proceedings of the Twentieth International Conference on Software Engineering*, April, Kyoto, 400-409.

Wohlin, C., Runesson, P., and Brantestam, J. (1995). "An Experimental Evaluation of Capture-Recapture in *Software Testing, Verification and Reliability*, 5, 213-232.

Capture - Recapture

Models, Methods, and the Reality

Jens-Peder Ekros
Anders Subotic

Bo Bergman

Linköping University, Sweden

Outline

- Uses of inspections
- Capture-recapture methods
- Test of models
- Results
- Conclusion & Future Work

Uses of inspections

- Primarily: *detection* of defects

- Additionally, inspection data can be used for:
  - Quality assurance, e.g. using *capture-recapture* methods
  - Quality control
  - Process improvement and organisational learning

- Additional uses of inspection data involves models of product, process and resource

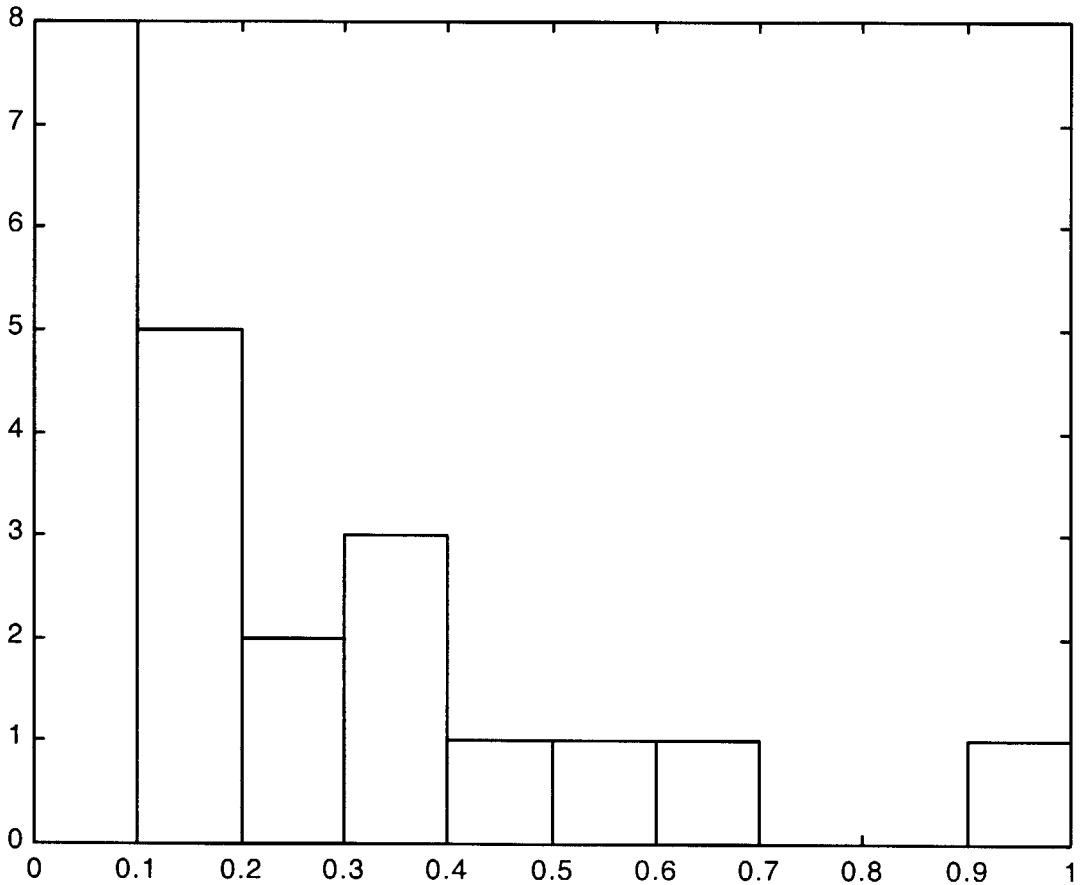
Capture-recapture methods

- Methods for estimating size of population
- Transferred from biology to software engineering
- Estimate defect content of software artefacts
- Common methods (estimators):
  - Jack-knife
  - Maximum likelihood
  - Chao
- Different *methods assume* different *models*, i.e. characteristics of input data

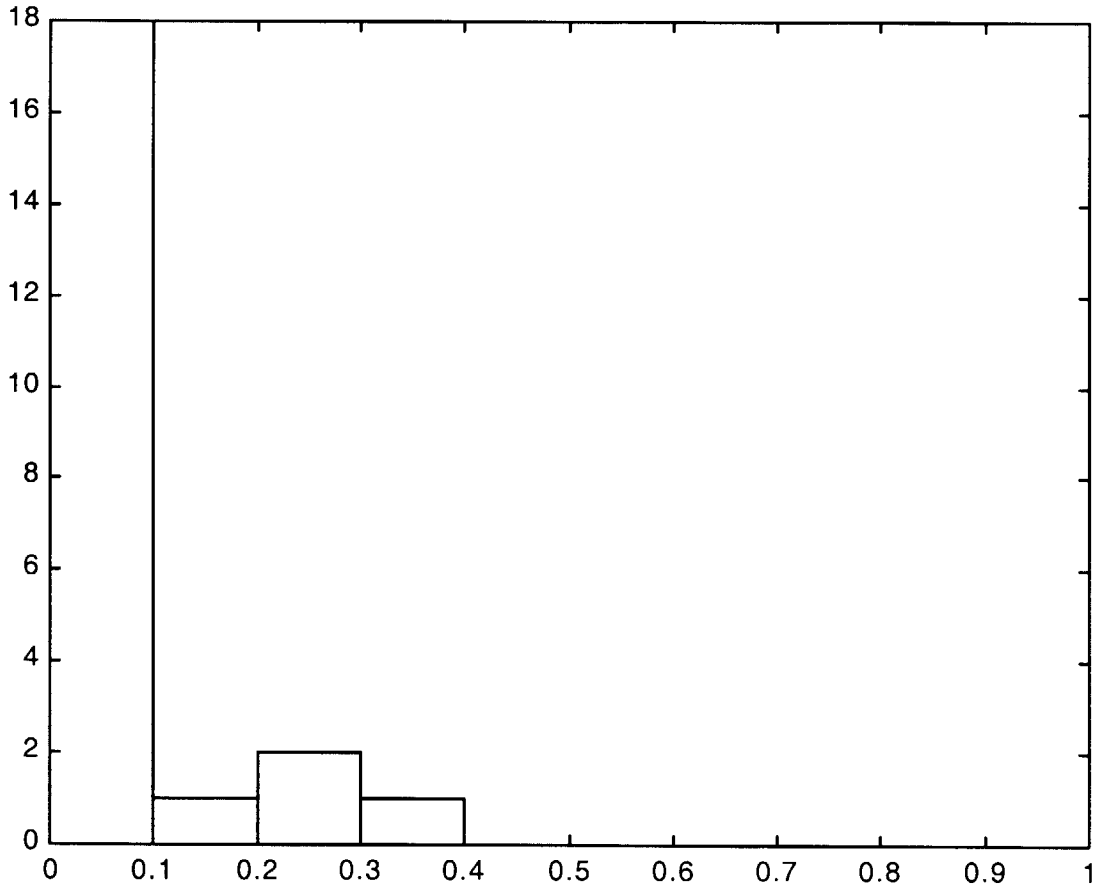
Test of models

- Three assumptions were tested:
  - Inspectors find the same number of defects (capability)
  - Defects are equally easy to detect (detectability)
  - Inspectors find the same defects
- Assumptions tested using old and new test statistics
- Distributions of test statistics created using Monte Carlo simulation and enumeration
- 24 published data sets were used (16 from NASA)

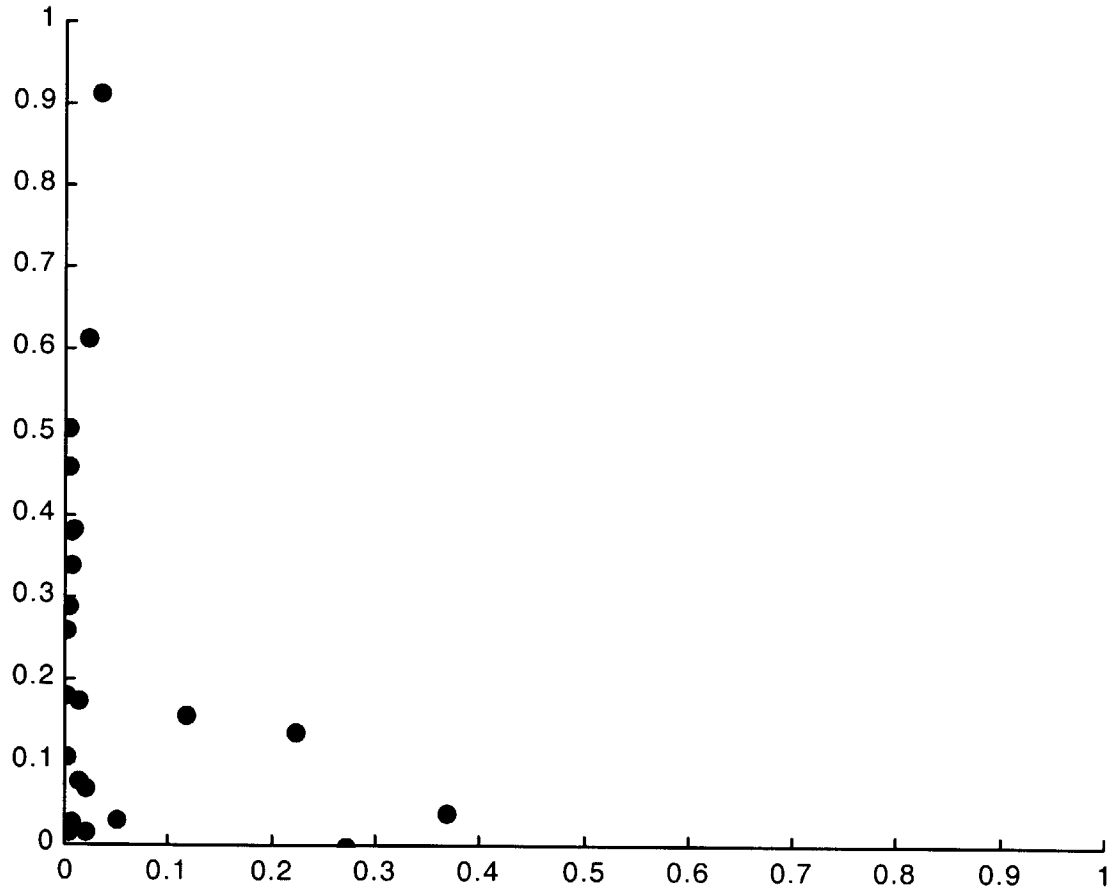
Inspector capability varies



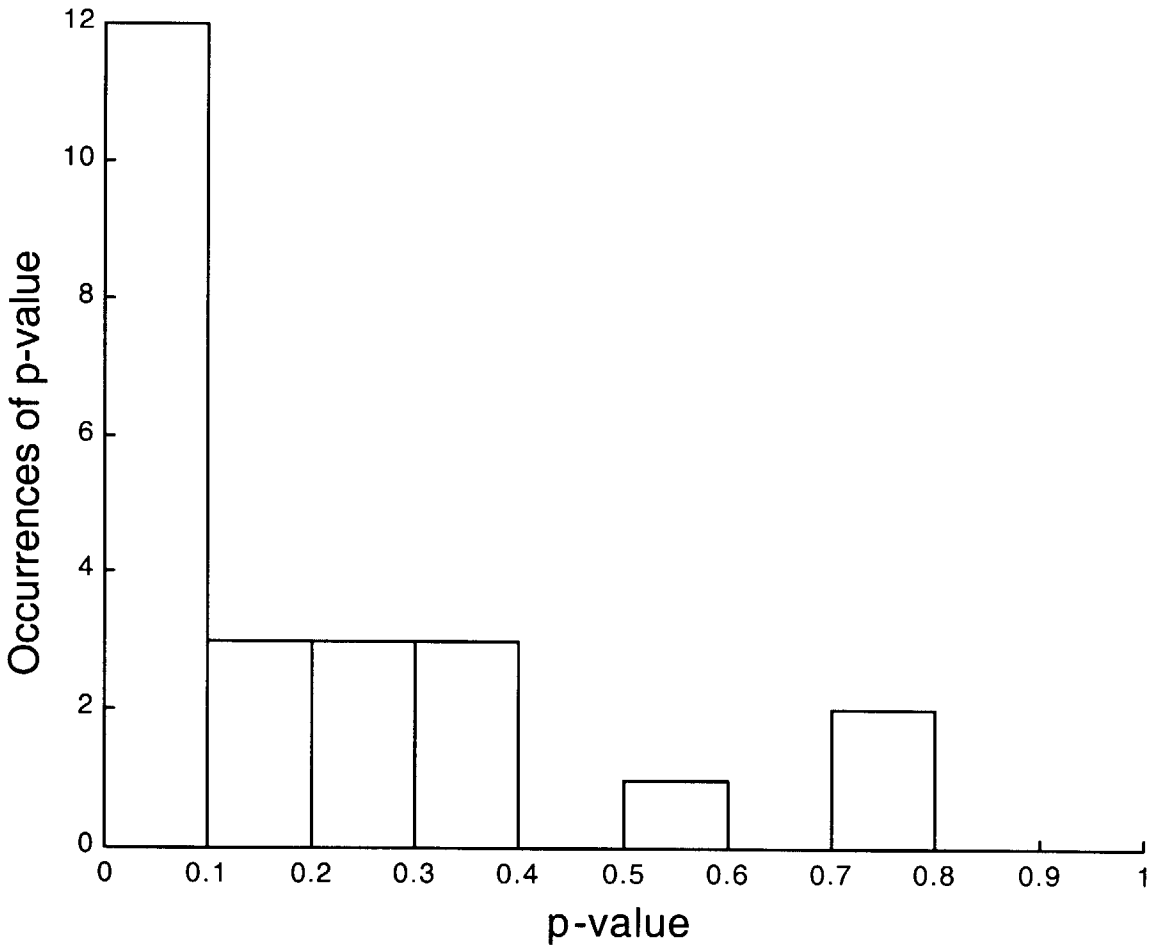
Defect detectability varies



Detectability varies more than capability



Inspectors do not find the same defects



Conclusion & Future work

- "Faults ain't fish" - Models have to be validated
- Tests using empirical data suggest:
  - Inspectors do not find the same number of defects
  - Defects are not equally easy to detect
  - Inspectors do not find the same defects

- How can we use these results for better estimates? Develop new or adjust existing estimation methods
- More industrial data involved in the analysis
- Investigate the effects of broken assumptions

Session 4: Fault Prediction

Software Evolution and the Fault Process
A. Nikora, Jet Propulsion Laboratory, and J. Munson, University of Idaho

Integrating Formal Methods Into Software Dependability Analysis
J. Knight and L. Nakano, University of Virginia

An Adaptive Software Reliability Prediction Approach
M. Yin, L. James, S. Keene, R. Arellano, and J. Peterson,
Raytheon Systems Company

SOFTWARE EVOLUTION AND THE FAULT PROCESS

Allen P. Nikora
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109-8099
Allen.P.Nikora@ jpl.nasa.gov

John C. Munson
Computer Science Department
University of Idaho
Moscow, ID 83844-1010
jmunson @cs.uidaho.edu

ABSTRACT

In developing a software system, we would like to estimate the way in which the fault content changes during its development, as well determine the locations having the highest concentration of faults. In the phases prior to test, however, there may be very little direct information regarding the number and location of faults. This lack of direct information requires developing a fault surrogate from which the number of faults and their location can be estimated. We develop a fault surrogate based on changes in the fault index, a synthetic measure which has been successfully used as a fault surrogate in previous work. We show that changes in the fault index can be used to estimate the rates at which faults are inserted into a system between successive revisions. We can then continuously monitor the total number of faults inserted into a system, the residual fault content, and identify those portions of a system requiring the application of additional fault detection and removal resources.

1. INTRODUCTION

Over a number of years of study, we can now establish a distinct relationship between software faults and certain aspects of software complexity. When a software system consisting of many distinct software modules is built for the first time, we have little or no direct information as to the location of faults in the code. Some of the modules will have far more faults in them then do others. We do, however, now know that the number of faults in a module is highly correlated with certain software attributes that may be measured. This means that we can measure the software on these specific attributes and have some reasonable notion as to the degree to which the modules are fault prone [Muns90, Muns96].

In the absence of information as to the specific location of software faults, we have successfully used a derived metric, the fault index measure, as a fault surrogate. That is, if the fault index of a module is large, then it will likely have a large number of latent faults. If, on the other hand, the fault index of a module is small, then it will tend to have fewer faults. As the software system evolves through a number of sequential builds, faults will be identified and the code will be changed in an attempt to eliminate the identified faults. The introduction of new code, however, is a fault prone process just as was the initial code generation. Faults may well be injected during this evolutionary process.

Code does not always change just to fix faults that have been isolated in it. Some changes to code during its evolution represent enhancements, design modifications or changes in the code in response to continually evolving requirements. These incremental code enhancements may also result in the introduction of still more faults. Thus, as a system progresses through a series of builds, the fault index of each program module that has been altered must also change. We will see that the rate of change in the system fault index will serve as a good index of the rate of fault introduction.

The general notion of software test is to make the rate of fault removal exceed the rate of fault introduction. In most cases, this is probably true [Muns97]. Some changes are rather more heroic than others. During these more substantive change cycles, it is quite possible that the actual number of faults in the system will rise. We would be very mistaken, then, to assume that software test will monotonically reduce the number of faults in a system. This will only be the case when the rate of fault removal exceeds the rate of fault introduction. The rate of fault removal is relatively easy to measure. The rate of fault introduction is much more tenuous. This fault introduction process is directly related to two measures that we can take on code as it evolves, fault deltas and net fault change (NFC).

In this investigation we establish a methodology whereby code can be measured from one build to the next, a measurement baseline. We use this measurement baseline to develop an assessment of the rate of change to a system as measured by our fault. From this change process we are then able to derive a direct measure of the rate of fault introduction based on changes in the software from one build to the next. Finally we examine data from an actual system on which faults may be traced to specific build increments to assess the predicted rate of fault introduction with the actual.

A major objective of this study is to identify a complete software system on which every version of every module has been archived together with the faults that have been recorded against the system as it evolved. For our purposes, the Cassini Orbiter Command and Data Subsystem at JPL met all of our objectives. On the first build of this system there were approximately 96K source lines of code in approximately 750 program modules. On the last build there were approximately 110K lines of source code in approximately 800 program modules. As the system progressed from the first to the last build there were a total of 45,200 different versions of these modules. On the average, then, each module progressed through an average of 60 evolutionary steps or versions. For the purposes of this study, the Ada program module is a procedure or function. it is the smallest unit of the Ada language structure that may be measured. A number of modules present in the first build of the system were removed on subsequent builds. Similarly, a number of modules were added.

The Cassini CDS does not represent an extraordinary software system. It is quite typical of the amount of change activity that will occur in the development of a system on the order of 100 KLOC. It is a non-trivial measurement problem to track the system as it evolves. Again, there are two different sets of measurement activities that must occur at once. We are interested the changes in the source code and we are interested in the fault reports that are being filed against each module.

2. A MEASUREMENT BASELINE

The measurement of an evolving software system through the shifting sands of time is not an easy task. Perhaps one of the most difficult issues relates to the establishment of a baseline against which the evolving systems may be compared. This problem is very similar to that encountered by the surveying profession. If we were to buy a piece of property, there are certain physical attributes that we would like to know about that property. Among these properties is the topology of the site. To establish the topological characteristics of the land, we will have to seek out a benchmark. This benchmark represents an arbitrary point somewhere on the subject property. The distance and the elevation of every other point on the property may then be established in relation to the measurement baseline. Interestingly enough, we can pick any point on the property, establish a new baseline, and get exactly the same topology for the property. The property does not change. Only our perspective changes.

When measuring software evolution, we need to establish a measurement baseline for this same purpose [Niko97, Muns96a]. We need a fixed point against which all others can be compared. Our measurement baseline also needs to maintain the property that, when another point is chosen, the exact same picture of software evolution emerges, only the perspective changes. The individual points involved in measuring software evolution are individual builds of the system.

For each raw metric in the baseline build, we may compute a mean and a standard deviation. Denote the vector of mean values for the baseline build as $\bar{\mathbf{x}}^B$ and the vector of standard deviations as $\mathbf{s}^B$. The standardized baseline metric values for any module j in an arbitrary build i, then, may be derived from raw metric values as

$$z_j^{B,i} = \frac{w_j^{B,i} - \bar{x}_j^B}{s_j^B}$$

Standardizing the raw metrics makes them more tractable. It now permits the comparison of metric values from one build to the next. From a software engineering perspective, there are simply too many metrics collected on each module over many builds. We need to reduce the dimensionality of the problem. We have successfully used principal components analysis for reducing the dimensionality of the problem [Muns90a, Khos92]. The principal components technique will reduce a set of highly correlated metrics to a much smaller set of uncorrelated or orthogonal measures. One of the products of the principal components technique is an orthogonal transformation matrix $\mathbf{T}$ that will send the standardized scores (the matrix $\mathbf{z}$) onto a reduced set of domain scores thusly, $\mathbf{d} = \mathbf{zT}$.

In the same manner as the baseline means and standard deviations were used to transform the raw metric of any build relative to a baseline build, the transformation matrix $\mathbf{T}^B$ derived from the baseline build will be used in subsequent builds to transform standardized metric values obtained from that build to the reduced set of domain metrics as follows: $\mathbf{d}^{B,i} = \mathbf{z}^{B,i}\,\mathbf{T}^B$, where $\mathbf{z}^{B,i}$ are the standardized metric values from build i baselined on build B.

Another artifact of the principal components analysis is the set of eigenvalues that are generated for each of the new principal components. Associated with each of the new measurement domains is an eigenvalue, λ. These eigenvalues are large or small varying directly with the proportion of variance explained by each principal component. We have successfully exploited these eigenvalues to create the fault index, ρ, that is the weighted sum of the domain metrics to wit: $\rho_i = 50 + 10\sum_{j=1}^{m} \lambda_j d_j$, where m is the dimensionality of the reduced metric set [Muns90a].

As was the case for the standardized metrics and the domain metrics, the fault index may be baselined as well, using the eigenvalues and the baselined domain values:

$$\rho_i^B = \sum_{j=1}^{m} \lambda_j^B d_j^B$$

If the raw metrics that are used to construct the fault index are carefully chosen for their relationship to software faults then the fault index will vary in exactly the same manner as the faults [Muns95]. The fault index is a very reliable fault surrogate. Whereas we cannot measure the faults in a program directly we can measure the fault index of the program modules that contain the faults. Those modules having a large fault index will ultimately be found to be those with the largest number of faults [Muns92].

3. SOFTWARE EVOLUTION

A software system consists of one or more software modules. As the system grows and modifications are made, the code is recompiled and a new version, or build, is created. Each build is constructed from a set of software modules. The new version may contain some of the same modules as the previous version, some entirely new modules and it may even omit some modules that were present in an earlier version. Of the modules that are common to both the old and new version, some may have undergone modification since the last build. When evaluating the change that occurs to the system between any two builds (software evolution), we are interested in three sets of modules. The first set, M_C, is the set of modules present in both builds of the system. These modules may have changed since the earlier version but were not removed. The second set, M_A, is the set of modules that were in the early build and were removed prior to the later build. The final set, M_B, is the set of modules that have been added to the system since the earlier build.

The fault index of the system R^i at build i, the early build, is given by

$$R^i = \sum_{c \in M_C} \rho_c^i + \sum_{a \in M_A} \rho_a^i .$$

Similarly, the fault index of the system R^j at build j, the later build is given by

$$R^j = \sum_{c \in M_C} \rho_c^j + \sum_{b \in M_B} \rho_b^j .$$

The later system build is said to be more fault prone if $R_j > R_i$.

As a system evolves through a series of builds, its fault burden will change. This burden may be estimated by a set of software metrics. One simple assessment of the size of a software system is the number of lines of code per module. However, using only one metric may neglect information about the other complexity attributes of the system, such as control flow and temporal complexity. By comparing successive builds on their domain metrics it is possible to see how these builds either increase or decrease based on particular attribute domains. Using the fault index, the overall system fault burden can be monitored as the system evolves.

Regardless of which metric is chosen, the goal is the same. We wish to assess how the system has changed, over time, with respect to that particular measurement. The concept of a code delta provides this information. A code delta is, as the name implies, the difference between two builds as to the relative complexity metric.

The change in the fault in a single module between two builds may be measured in one of two distinct ways. First, we may simply compute the simple difference in the module fault index between build *i* and build *j*. We have called this value the fault delta for the module *m*, or $\delta_m^{i,j} = \rho_m^j - \rho_m^i$. A limitation of measuring fault deltas is that it doesn't give an indicator as to how much change the system has undergone. If, between builds, several software modules are removed and are replaced by modules of roughly equivalent complexity, the fault delta for the system will be close to zero. The overall complexity of the system, based on the metric used to compute deltas, will not have changed much. However, the reliability of the system could have been severely affected by the replacing old modules with new ones. What we need is a measure to accompany fault delta that indicates how much change has occurred.

The absolute value of the fault delta is a measure of code churn. In the case of code churn, what is important is the absolute measure of the nature that code has been modified. From the standpoint of fault insertion, removing a lot of code is probably as catastrophic as adding a bunch. The new measure of net fault change (NFC), χ, for module *m* is simply

$$\chi_m^{i,j} = \left|\delta_m^{i,j}\right| = \left|\rho_m^i - \rho_{mJ}\right|$$

The total change of the system is the sum of the fault delta's for a system between two builds *i* and *j* is given by

$$\Delta^{i,j} = \sum_{c \in M_C} \delta_c^{i,j} - \sum_{a \in M_A} \rho_a^i + \sum_{b \in M_B} \rho_b^j .$$

Similarly, the NFC of the same system over the same builds is

$$\nabla^{i,j} = \sum_{c \in M_C} \chi_c^{i,j} + \sum_{a \in M_A} \rho_a^i + \sum_{b \in M_B} \rho_b^j .$$

With a suitable baseline in place, and the module sets defined above, it is now possible to measure software evolution across a full spectrum of software metrics. We can do this first by comparing average metric values for the different builds. Secondly, we can measure the increase or decrease in system complexity as measured by a selected metric, fault delta, or we can

measure the total amount of change the system has undergone between builds, net fault change.

4. OBTAINING AVERAGE BUILD VALUES

One synthetic software measure, fault index, has clearly been established as a successful surrogate measure of software faults [Muns90a]. It seems only reasonable that we should use it as the measure against which we compare different builds. Since the fault index is a composite measure based on the raw measurements, it incorporates the information represented by *LOC*, *V(g)*, η_1, η_2, and all the other raw metrics of interest. The fault index is a single value that is representative of the complexity of the system which incorporates all of the software attributes we have measured (e.g. size, control flow, style, data structures, etc.).

By definition, the average fault index, $\bar{\rho}$, of the baseline *system* will be

$$\bar{\rho}^B = \frac{1}{N^B}\sum_{i=1}^{N^B} \rho_i^B = 50,$$

where N^B is the cardinality of the set of modules on build *B*, the baseline build. The fault index for the baseline build is calculated from standardized values using the mean and standard deviation from the baseline metrics. The fault indices are then scaled to have a mean of 50 and a standard deviation of 10. For that reason, the average fault index for the baseline system will always be a fixed point. Subsequent builds are standardized using the means and standard deviations of the metrics gathered from the baseline system to allow comparisons. The average fault index for subsequent builds is given by

$$\bar{\rho}^k = \frac{1}{N^k}\sum_{i=1}^{N^k} \rho_i^{B,k},$$

where N^k is the cardinality of the set of program modules in the k^{th} build and $\rho_i^{B,k}$ is the baselined fault index for the i^{th} module of that set.

As the code is modified over time, faults will be found and fixed. However, new faults will be introduced into the code as a result of the change. In fact, this fault introduction process is directly proportional to change in the program modules from one version to the next. As a module is changed from one build to the next in response to evolving requirements changes and fault reports, its measurable software attributes will also change. Generally, the net effect of a change is that complexity will increase. Only rarely will its complexity decrease.

5. DEFINITION OF A FAULT

Unfortunately there is no particular definition of precisely what a software fault is. This makes it difficult to develop meaningful associative models between faults and metrics. In calibrating our model, we would like to know how to count faults in an accurate and repeatable manner. In measuring the evolution of the system to talk about rates of fault introduction and removal, we measure in units to the way that the system changes over time. Changes to the system are visible at the module level, and we attempt to measure at that level of granularity. Since the measurements of system structure are collected at the module level (by module we mean procedures and functions), we would like information about faults at the same granularity. We would also like to know if there are quantities that are related to fault counts that can be used to make our calibration task easier.

Following the second definition of fault in [IEEE83, IEEE88], we consider a fault to be a **structural imperfection** in a software system that **may** lead to the system's eventually failing. In other words, it is a **physical characteristic** of the system of which the type and extent may be measured using the same ideas used to measure the properties of more traditional physical systems. Faults are introduced into a system by people making errors in their tasks - these errors may be errors of commission or errors of omission. In order to count faults, we needed to develop a method of identification that is repeatable, consistent, and identifies faults at the same level of granularity as our structural measurements. Faults may be local – for instance, a system might contain an implementation fault affecting only one module in which the programmer incorrectly initializes a variable local to the routine. Faults may also span multiple modules - for instance, each module containing an include file with a particular fault would have that fault. In identifying and counting faults, we must deal with both types of faults. Details of the fault counting and identification rules developed for this study are given in [Niko97a, Niko98]

In analyzing the flight software for the CASSINI project the fault data and the source code change data were available from two different systems. The problem reporting information was obtained from the JPL institutional problem reporting system. Failures were recorded in this system starting at subsystem-level integration, and continuing through spacecraft integration and test. Failure reports typically contain descriptions of the failure at varying levels of detail, as well as descriptions of what was done to correct the fault(s) that caused the failure. Detailed information regarding the underlying faults (e.g., where were the code changes made in each affected module) is generally unavailable from the problem reporting system.

The entire source code evolution history could be obtained directly from the Software Configuration Control System (SCCS) files for all versions of the flight software. The way in which SCCS was used in this development effort makes it possible to track changes to

the system at a module level in that each SCCS file stores the baseline version of that file (which may contain one or more modules) as well as the changes required to produce each subsequent increment (SCCS delta) of that file. When a module was created, or changed in response to a failure report or engineering change request, the file in which the module is contained was checked into SCCS as a new delta. This allowed us to track changes to the system at the module level as it evolved over time. For approximately 10% of the failure reports, we were able to identify the source file increment in which the fault(s) associated with a particular failure report were repaired. This information was available either in the comments inserted by the developer into the SCCS file as part of the check-in process, or as part of the set of comments at the beginning of a module that track its development history.

Using the information described above, we performed the following steps to identify faults. First, for each problem report, we searched all of the SCCS files to identify all modules and the increment(s) of each module for which the software was changed in response to the problem report. Second, for each increment of each module identified in the previous step, we assumed as a starting point that all differences between the increment in which repairs are implemented and the previous increment are due solely to fault repair. Note that this is not necessarily a valid assumption - developers may be making functional enhancements to the system in the same increment that fault repairs are being made. Careful analysis of failure reports for which there was sufficiently detailed descriptive information served to separate areas of fault repair from other changes. However, the level of detail required to perform this analysis was not consistently available. Third, we used a differential comparator (e.g., Unix `diff`) to obtain the differences between the increment(s) in which the fault(s) were repaired, and the immediately preceding increment(s). The results indicated the areas to be searched for faults.

After completing the last step, we still had to identify and count the faults - the results of the differential comparison cannot simply be counted up to give a total number of faults. In order to do this, we developed a taxonomy for identifying and counting faults [Niko98]. This taxonomy differs from others in that it does not seek to identify the root cause of the fault. Rather, it is based on the types of changes made to the software to repair the faults associated with failure reports - in other words, it constitutes an operational definition of a fault. Although identifying the root causes of faults is important in improving the development process [Chil92, IEEE93], it is first necessary to identify the faults. We do not claim that this is the only way to identify and count faults, nor do we claim that this taxonomy is complete. However, we found that this taxonomy allowed us to successfully identify faults in the software used in the study in a consistent manner at the appropriate level of granularity.

6. THE RELATIONSHIP BETWEEN FAULTS AND CODE CHANGES

Having established a theoretical relationship between software faults and code changes, it is now of interest to validate this model empirically. This measurement occurred on two simultaneous fronts. First, all of the versions of all of the source code modules were measured. From these measurements, NFC and fault deltas were obtained for every version of every module. The failure reports were sampled to lead to specific faults in the code. These faults were classified according to the above taxonomy manually on a case by case basis. Then we were able to build a regression model relating the code measures to the code faults.

The Ada source code modules for all versions of each of these modules were systematically reconstructed from the SCCS code deltas. Each of these module versions was then measured by the UX-Metric analysis tool for Ada [SETL93]. Not all metrics provided by this tool were used in this study. Only a subset of these actually provide distinct sources of variation [Khos90]. The specific metrics used in this study are shown in Table 1.

Metrics	Definition
η_1	Count of unique operators [Hal77]
η_2	Count of unique operands
N_1	Count of total operators
N_2	Count of total operands
P/R	Purity ratio: ratio of Halstead's $\hat{N}$ to total program vocabulary
V(g)	McCabe's cyclomatic complexity
Depth	Maximum nesting level of program blocks
AveDepth	Average nesting level of program blocks
LOC	Number of lines of code
Blk	Number of blank lines
Cmt	Count of comments
CmtWds	Total words used in all comments
Stmts	Count of executable statements
LSS	Number of logical source statements
PSS	Number of physical source statements
NonEx	Number of non-executable statements
AveSpan	Average number of lines of code between references to each variable
VI	Average variable name length

Table 1. Software Metric Definitions

To establish a baseline system, all of the metric data for the module versions that were members of the first build of CDS were then analyzed by our PCA-FI tool. This tool is designed to compute fault indices either from a baseline system or from a system being compared to

the baseline system. In that the first build of the Cassini CDS system was selected to be the baseline system, the PCA-FI tool performed a principal components analysis on these data with an orthogonal varimax rotation. The objective of this phase of the analysis is to use the principal components technique to reduce the dimensionality of the metric set. As may been seen in Table 2, there are four principal components for the 18 metrics shown in Table 1. For convenience, we have chosen to name these principal components as **Size**, **Structure**, **Style** and **Nesting**. From the last row in Table 2 we can see that the new reduced set of orthogonal components of the original 18 metrics account for approximately 85% of the variation in the original metric set.

Metric	Size	Structure	Style	Nesting
Stmts	0.968	0.022	-0.079	0.021
LSS	0.961	0.025	-0.080	0.004
N_2	**0.926**	0.016	0.086	0.086
N_1	**0.934**	0.016	0.074	0.077
η_2	**0.884**	0.012	-0.244	0.043
AveSpan	**0.852**	0.032	0.031	-0.082
V(g)	**0.843**	0.032	-0.094	-0.114
η_1	**0.635**	-0.055	-0.522	-0.136
Depth	**0.617**	-0.022	-0.337	-0.379
LOC	-0.027	**0.979**	0.136	0.015
Cmt	-0.046	**0.970**	0.108	0.004
PSS	-0.043	**0.961**	0.149	0.019
CmtWds	0.033	**0.931**	0.058	-0.010
NonEx	-0.053	**0.928**	0.076	-0.009
Blk	0.263	**0.898**	0.048	0.005
P/R	-0.148	-0.198	**-0.878**	0.052
Vl	0.372	-0.232	**-0.752**	0.010
AveDepth	-0.000	-0.009	0.041	**-0.938**
% Variance	37.956	30.315	10.454	6.009

Table 2. Principal Components of Software Metrics

As is typical in the principal components analysis of metric data, the **Size** domain dominates the analysis. It alone accounts for approximately 38% of the total variation in the original metric set. Not surprisingly, this domain contains the metrics of total statement count (*Stmts*), logical source statements (LSS), the Halstead lexical metric primitives of operator and operand count, but it also contains cyclomatic complexity (*V(g)*). In that we regularly find cyclomatic complexity in this domain we are forced to conclude that it is only a simple measure of size in the same manner as statement count. The **Structure** domain contain those metrics relating to the physical structure of the program such as non-executable statements (*NonEx*) and the program block count (*Blk*). The **Style** domain contains measures of attribute that are directly under a programmer's control such as variable length (*Vl*) and purity ratio (*P/R*). The **Nesting** domain consist of the single metric that is a measure of the average depth of nesting of program modules (*AveDepth*).

In order to transform the raw metrics for each module version into their corresponding fault indices, the means and the standard deviations must be computed. These values will be used to transform all raw metric values for all versions of all modules to their baselined *z* score values. The transformation matrix will then map the metric *z* score values onto their orthogonal equivalents to obtain the orthogonal domain metric values used in the computation of the fault index. With this information, we can obtain baselined fault index values for any version of any module relative to the baseline build. As an aside, it is not necessary that the baseline build be the initial build. As a typical system progresses through hundreds of builds in the course of its life, it is worth reestablishing a baseline closer to the current system. In any event, these baseline data are saved by the PCA-FI tool for use in later computation of metric values. Whenever the tool is invoked referencing the baseline data it will automatically use these data to transform the raw metric values given to it.

Once the baselined fault index data have been assembled for all versions of all modules, it is then possible to examine some trends that have occurred during the evolution of the system. For example, in Figure 1 the fault index of the evolving CDS system is shown across one of its five major builds. To compute these changing fault index values, every development increment within that build was identified. Then, for each increment, the baselined fault indices of the modules in that increment were computed. The next four increments, not shown here, have evolutionary patterns similar to that shown in Figure 1. It seems to be that the average fault index of most systems is a monotonically increasing function.

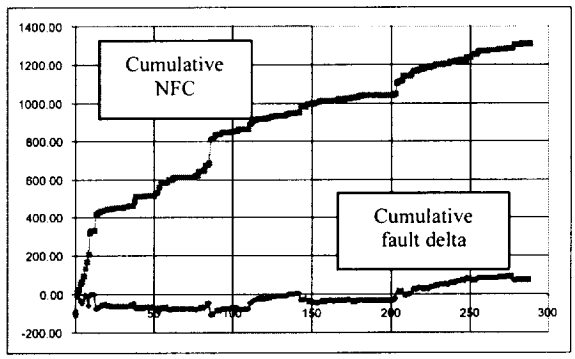


Figure 1. Change in the Fault Index for One Version of CDS Flight Software

Note in Figure 1 that not all increments within a build represent the same increase in the fault index. Nearly one third of the total change in this version takes place within the first 10% of the development increments. From our understanding of the relationship between the fault index and injected faults, we would expect that the magnitude of change within the first 30 increments would indicate that a large number of faults

would have been injected as a result of this activity. It is also interesting to note that the final fault index of this particular version is rather close to the initial fault index, although it is quite clear from the measured activity that a significant amount of change has occurred.

Not all program modules received the same degree of modification as the system evolved. Some modules changed relatively little. Figure 2 shows the net fault change and fault delta values for a module that was relatively stable over its change history. There were only four relatively minor changes to this module. A more typical change history is shown for another module in Figure 3. The total net fault change for this module is approximately 38. It is interesting to note that the fault delta for this module is close to zero. The fault index of the module at the last version is very close to its original value. This figure clearly illustrates the conceptual differences between the two measure of net fault change and fault delta.

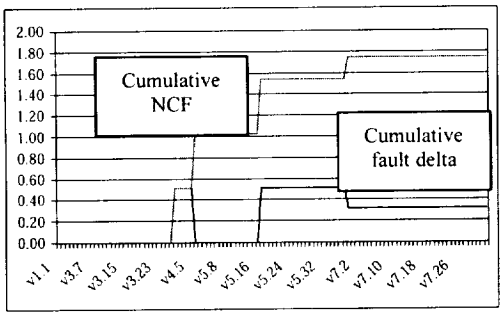


Figure 2. Change History for Stable Module

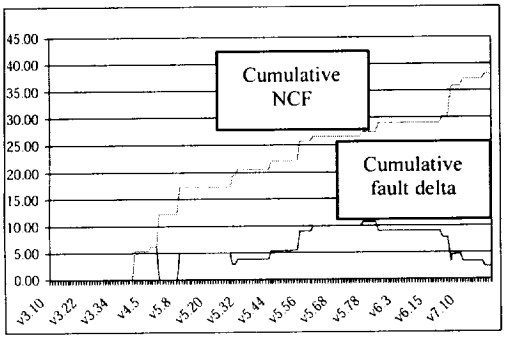


Figure 3. Typical Module Change History

Figure 4 shows a module at the extreme end of change history. This module has a total net fault change value of close to 140. Also, its final fault delta value is about 30, indicating that its fault index has also increased significantly as it evolved. Among the three modules whose change history is illustrated by Figures 2, 3, and 4, the latter module is the one that we focus our attention on the most. It is the one most likely to have had significant numbers of faults introduced into it throughout its dramatic life.

Now let us turn our attention to the fault identification process. Over 600 failure reports were written against the CDS flight software during developmental testing and system integration. Failure reports contain a description of how the system's behavior deviated from expectations, the date on which the failure was observed, and a description of the corrective action that was taken.

In relating the number of faults inserted in an increment to measures of a module's structural change, we had only a small number of observations with which to work. There were three difficulties that had to be dealt with. First, recall that for only about 10% of the failure reports were we able to identify the module(s) that had been changed, and in which increment those changes were made. Although the development practices used on this project included the placement of comments in the source code to identify repair activities resulting from each problem report, this requirement was not consistently enforced. Second, once a fault had been identified, it was necessary to trace it back to the increment in which it first occurred. For some source files, there were over 100 increments that had to be manually searched. Since the SCCS files for each delivered version were available, it was possible to trace most faults back to their point of origin. As previously noted, the principal difficulty was the sheer volume of material that had to be examined – this was one of the factors restricting the number of observations that could be obtained. Third, there were numerous instances in which the UX-Metric analyzer that was used to obtain the raw structural measurements would not measure a particular module. The net result was that of the over 100 faults that were initially identified, there were only 35 observations in which a fault could be associated with a particular increment of a module, and with that increment's measures of fault delta and net fault change.

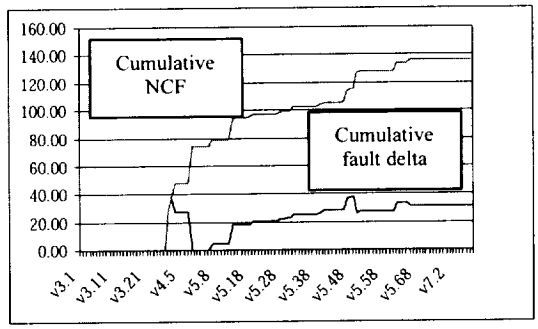


Figure 4. Change History for Frequently Changed Module

For each of the 35 modules for which there was viable fault data, there were three data points. First, we had the number of injected faults for that module that were the direct result of changes that had occurred on that module between the current version that contained the faults and the previous version that did not. Second, we had fault delta values for each of these modules from

the current to the previous version. Finally, we had net fault change values derived from the fault deltas.

Linear regression models were computed for net fault change and fault deltas with actual code faults as the dependent variable in both cases. Both models were build without constant terms in that we surmise that if no changes were made to a module, then no new faults could be introduced. The results of the regression between faults and fault deltas were not at all surprising. The squared multiple R for this model was 0.001, about as close to zero as you can get. This result is directly attributable to the non-linearity of the data. Change comes in two flavors. Change may increase the complexity of a module. Change may decrease the complexity of a model. Faults, on the other hand are not related to the direction of the change but to its intensity. Removing masses of code from a module is just as likely to introduce faults and adding code to it.

The regression model between net fault change and faults is dramatically different. The regression ANOVA for this model are shown in Table 3. Whereas fault deltas do not show a linear relationship with faults, net fault change certainly does. The actual regression model is given in Table 4. In Table 5 the regressions statistics have been reported. Of particular interest is the Squared Multiple R term, having a value of 0.653. This means, roughly, that the regression model will account for more than 65% of the variation in the faults of the observed modules based on the values of net fault change.

Source	Sum-of-Squares	DF	Mean-Square	F-Ratio	P
Regression	331.879	1	331.879	62.996	0.000
Residual	179.121	34	10.673	5.268	

Table 3. Regression Analysis of Variance

Effect	Coefficient	Std Err	t	P(2-Tail)
NFC	0.576	0.073	7.937	0.000

Table 4. Regression Model

N	Multiple R	Squared multiple R	Standard error of estimate
35	0.806	0.649	2.296

Table 5. Regression Statistics

Of course, it may be the case that both the amount of change and the direction in which the change occurred. The linear regression through the origin shown in Tables 6, 7, and 8 below illustrates this model.

Source	Sum-of-Squares	DF	Mean-Square	F-Ratio	P
Regression	367.247	2	183.623	42.153	0.000
Residual	143.753	33	4.356		

Table 6. Regression Analysis of Variance

Effect	Coefficient	Std Err	t	P(2-Tail)
NFC	0.647	0.071	9.172	0.000
Delta	0.201	0.071	2.849	0.002

Table 7. Regression Model

N	Multiple R	Squared multiple R	Standard error of estimate
35	.848	.719	2.087

Table 8. Regression Statistics

We see that the model incorporating fault delta as well as net fault change performs significantly better than the model incorporating net fault change alone, as measured by Squared Multiple R and Mean Sum of Squares.

We determined whether the linear regression model which uses net fault change alone is an adequate predictor at a particular significance level when compared to the model using both net fault change and fault delta. We used the R^2-adequate test [MacD97, Net83] to examine the linear regression models through the origin and determine whether the models that depend only on structural measures are an adequate predictor. A subset of predictor variables is said to be R^2-adequate at significance level α if:

$R^2_{sub} > 1 - (1 - R^2_{full})(1 + d_{n,k})$, where

- R^2_{sub} is the R^2 value achieved with the subset of predictors
- R^2_{full} is the R^2 value achieved with the full set of predictors
- $d_{n,k} = (kF_{k,n-k-1})/n\text{-}k\text{-}1$, where
  - k = number of predictor variables in the model
  - n = number of observations
  - F = F statistic for significance α for n,k degrees of freedom.

Table 9 below show values of R^2, k, degrees of freedom, $F_{k,n-k-1}$, $d_{n,k}$, and R^2_{sub} for all four linear regression models through the origin. The number of observations, n, is 35, and we specify a value of α=.05.

We see in Table 9 that the value of Multiple Squared R for the regression using only net fault change is 0.649, and the 5% significance threshold for the net fault change and fault delta regression model is 0.661. This means that the regression model using only NFC is not R^2 adequate when compared to the model using both net fault change and fault delta as predictors. The amount of change occurring between subsequent revisions and the direction of that change both appear to be important in determining the number of faults inserted into a system.

Lin. Regressions Through Origin	R^2	DF	k	$F_{k,n-k-1}$ for significance α	d(n,k)	Threshold for significance α
NFC only	0.649	34	1	4.139	0.125	-----
NFC, Fault Delta	0.719	33	2	3.295	0.206	0.661

Table 9. Values of R^2, DOF, k, $F_{k,n-k-1}$, and $d_{n,k}$ for R^2-adequate Test

Finally, we examined the predicted residuals for the linear regression models described above. Table 10 be-

low shows the results of the Wilcoxon Signed Ranks test, as applied to the predictions for the excluded observations and the number of faults observed for each of the two linear regression models through the origin. For these models, about 2/3 of the estimates tend to be less than the number of faults observed.

Plots of the predicted residuals against the actual number of observed faults for each of the linear regression models through the origin are shown in Figures 5 and 6 below. The results of the Wilcoxon signed ranks tests, as well as Figures 5 and 6, indicate that the predictive accuracy of the regression models might be improved if syntactic analyzers capable of measuring additional aspects of a software system's structure were available. Recall, for instance, that we did not measure any of the real-time aspects of the system. Analyzers capable of measuring changes in variable definition and usage as well changes to the sequencing of blocks might also provide more accurate measurements.

Sample Pair		N	Mean Rank	Sum of Ranks	Test Statistic Z	Asymptotic Significance (2-tailed)
Observed Faults; NFC only fault est.	Neg.	25[a]	17.52	438.00	-2.015[d]	.044
	Pos.	10[b]	19.20	192.00		
	Ties	0[c]				
	Total	35				
Observed Faults; NFC and Fault Delta est.	Neg.	24[a]	16.92	406.00	-1.491[d]	.136
	Pos.	11[b]	20.36	224.00		
	Ties	0[c]				
	Total	35				

a. Observed Faults > Regression model predictions
b. Observed Faults < Regression model predictions
c. Observed Faults = Regression model predictions
d. Based on positive ranks

Table 10. Wilcoxon Signed Ranks Test for Linear Regressions Through the Origin

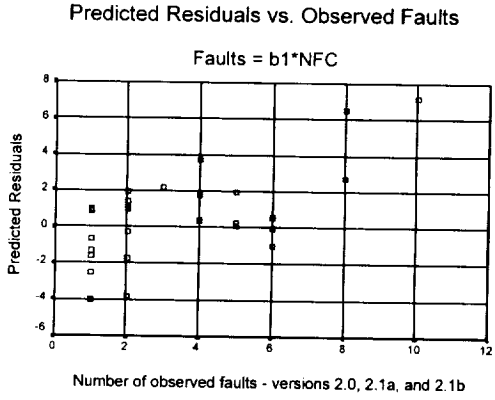


Figure 5. Predicted Residuals vs. Number of Observed Faults for Linear Regression Using NFC

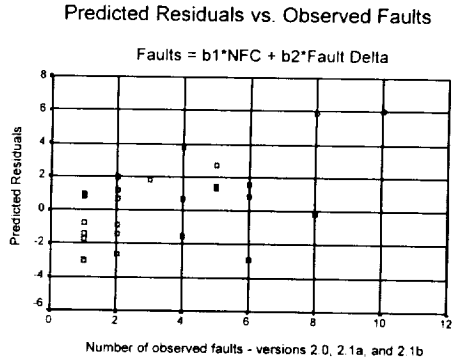


Figure 6. Predicted Residuals vs. Number of Observed Faults for Linear Regression with NFC and Fault Delta

7. SUMMARY

There is a distinct and a strong relationship between software faults and measurable software attributes. This is in itself not a new result or observation. The most interesting result of this endeavor is that we also found a strong association between the fault introduction process over the evolutionary history of a software system and the degree of change taking place in each of the program modules. We also found that the direction of the change was significant in determining the number of faults inserted. Some changes will have the potential of introducing very few faults while others may have a serious impact on the number of latent faults. Different numbers of faults may be inserted, depending upon whether code is being added to or removed from the system.

In order for the measurement process to be meaningful, fault data must be very carefully collected. In this study, the data were extracted ex post facto as a very labor intensive effort. Since fault data cannot be collected with the same degree of automation as much of the data on software metrics being gathered by development organizations, material changes in the software development and software maintenance processes must be made to capture these fault data. Among other things, a well defined fault standard and fault taxonomy must be developed and maintained as part of the software development process. Further, all designers and coders should be trained in its use. A viable standard is one that may be used to classify any fault unambiguously. A viable fault recording process is one in which any one person will classify a fault exactly the same as any other person.

Finally, the whole notion of measuring the fault introduction process is its ultimate value as a measure of software process. The software engineering literature is replete with examples of how software process improvement can be achieved through the use of some new software development technique. What is almost absent from the same literature is a controlled study to validate

the fact that the new process is meaningful. The techniques developed in this study can be implemented in a development organization to provide a consistent method of measuring fault content and structural evolution across multiple projects over time. We are working with software development efforts at JPL to address the practical aspects of inserting these measurement techniques into production software development environments. The initial estimates of fault insertion rates can serve as a baseline against which future projects can be compared to determine whether progress is being made in reducing the fault insertion rate, and to identify those development techniques that seem to provide the greatest reduction.

ACKNOWLEDGMENTS

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

[Chil92] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M.-Y. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurement", IEEE Transactions on Software Engineering, November, 1992, pp. 943-946.

[Hal77] M. H. Halstead, *Elements of Software Science.* Elsevier, New York, 1977.

[IEEE83] "IEEE Standard Glossary of Software Engineering Terminology", IEEE Std 729-1983, Institute of Electrical and Electronics Engineers, 1983.

[IEEE88] "IEEE Standard Dictionary of Measures to Produce Reliable Software", IEEE Std 982.1-1988, Institute of Electrical and Electronics Engineers, 1989.

[IEEE93] "IEEE Standard Classification for Software Anomalies", IEEE Std 1044-1993, Institute of Electrical and Electronics Engineers, 1994

[Khos90] T. M. Khoshgoftaar and J. C. Munson , "Predicting Software Development Errors Using Complexity Metrics," *IEEE Journal on Selected Areas in Communications* 8, 1990, pp. 253-261.

[Khos92] T. M. Khoshgoftaar and J. C. Munson "A Measure of Software System Complexity and Its Relationship to Faults," In *Proceedings of the 1992 International Simulation Technology Conference*, The Society for Computer Simulation, San Diego, CA, 1992, pp. 267-272.

[MacD97] S. G. MacDonell, M. J. Shepperd, P. J. Sallis, "Metrics for Database Systems: An Empirical Study", Proceedings of the Fourth International Software Metrics Symposium, November 5-7, 1997, Albuquerque, NM, pp. 99-107

[Muns90] J. C. Munson and T. M. Khoshgoftaar "Regression Modeling of Software Quality: An Empirical Investigation," *Journal of Information and Software Technology*, 32, 1990, pp. 105-114.

[Muns90a] J. C. Munson and T. M. Khoshgoftaar "The Relative Software Complexity Metric: A Validation Study," In *Proceedings of the Software Engineering 1990 Conference*, Cambridge University Press, Cambridge, UK, 1990, pp. 89-102.

[Muns92] J. C. Munson and T. M. Khoshgoftaar "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering*, SE-18, No. 5, 1992, pp. 423-433.

[Muns95] J. C. Munson, "Software Measurement: Problems and Practice," *Annals of Software Engineering*, J. C. Baltzer AG, Amsterdam 1995.

[Muns96] J. C. Munson, "Software Faults, Software Failures, and Software Reliability Modeling", *Information and Software Technology*, December, 1996.

[Muns96a] J. C. Munson and D. S. Werries, "Measuring Software Evolution," *Proceedings of the 1996 IEEE International Software Metrics Symposium* , IEEE Computer Society Press, pp. 41-51.

[Muns97] J. C. Munson and G. A. Hall, "Estimating Test Effectiveness with Dynamic Complexity Measurement," *Empirical Software Engineering Journal.* Feb. 1997.

[Net83] J. Neter, W. Wasserman, M. H. Kutner, Applied Linear Regression Models, Irwin: Homewood, IL, 1983

[Niko97] A. P. Nikora, N. F. Schneidewind, J. C. Munson, "IV&V Issues in Achieving High Reliability and Safety in Critical Control System Software", proceedings of the International Society of Science and Applied Technology conference, March 10-12, 1997, Anaheim, CA, pp 25-30.

[Niko97a] A. P. Nikora, J. C. Munson, "Finding Fault with Faults: A Case Study", proceedings of the Annual Oregon Workshop on Software Metrics, Coeur d'Alene, ID, May 11-13, 1997

[Niko98] A. P. Nikora, "Software System Defect Content Prediction From Development Process And Product Characteristics", Doctoral Dissertation, Department of Computer Science, University of Southern California, May, 1998.

[SETL93] "User's Guide for UX-Metric 4.0 for Ada", SET Laboratories, Mulino, OR, © SET Laboratories, 1987-1993

11
IN-61

INTEGRATING FORMAL METHODS INTO SOFTWARE DEPENDABILITY ANALYSIS

John C. Knight Luís G. Nakano

(knight | nakano)@virginia.edu

Department of Computer Science
University of Virginia
Charlottesville, VA 22903-2442, USA

An abstract submitted to:

The Twenty-Third Goddard Software Engineering Laboratory Workshop

Contact author:

John C. Knight

Department of Computer Science
University of Virginia
Thornton Hall
Charlottesville, VA 22903-2442, USA

knight@virginia.edu
+1 804 982 2216 (Voice)
+1 804 982 2214 (FAX)

INTEGRATING FORMAL METHODS INTO SOFTWARE DEPENDABILITY ANALYSIS

John C. Knight
Department of Computer Science
University of Virginia

Luís G. Nakano
Department of Computer Science
University of Virginia

1. Introduction

Formal methods are techniques based in mathematics that facilitate the precise specification and verification of software systems. Their use has been demonstrated in a number of experiments and industrial development projects [3]. Despite these demonstrations, formal techniques remain the exception rather than the rule in system development. One of the issues raised about the use of formal methods is the lack of any means whereby their results can be used in the broader context of system dependability analysis, i.e., the analysis of a complete system including hardware and software. For example, what would be the benefit at the *system* level of the use of a formal specification in the preparation of the system *software*?

The issues that we address in the work summarized here are:

- For what parts of a complex software system should formal methods be used?
- How can the results of formal analysis be used in the overall dependability analysis of the entire system?

We summarize a process by which these issues are addressed, and show thereby how to determine the role of formal methods in any particular development and how to exploit the results of formal analysis in system dependability analysis. At the workshop, we will illustrate the process using examples from analysis performed on parts of the design of an experimental nuclear-reactor control system.

2. Dependability Analysis

Analysis of the dependability of safety-critical systems is essential in order to provide estimates of the expected losses (life and/or property) that such systems will cause per unit of operating time, i.e., their risks. These risk estimates are used by developers, users, policy makers and others to make informed decisions about deploying safety-critical systems based on the expected benefits and losses to society.

Risk analysis has not been applied as successfully to software-based safety-critical systems as it has to hardware-only systems. The reason is the discrete nature of software—it causes complexity not usually found in analog hardware and prevents interpolation of test results commonly applied to hardware-only systems. The result is a situation in which the hardware elements of a system are typically analyzed in depth but software is handled in only a very limited way, often as a "black box".

Life testing is an approach to software dependability assessment in which the software is treated as a black box. The software is executed continuously in its operating environment for a period of time proportional to the duration of the mission and inversely proportional to the acceptable probability of failure. Unfortunately, it has been shown [2] that life testing is not a feasible approach to the dependability assessment of life-critical software because the duration of testing required is excessive. To reduce the need for testing, reliability growth models have also been tried [1]. By modeling the development of software in terms of testing and fault removal, it is argued that an estimate for software reliability can be obtained with lower test requirements. If it works at all, this approach only works for modest levels of dependability.

Formal methods are often advocated as an approach to developing dependable software. But poor tool support, the complexity of the systems, and the difficulty of using the techniques have limited the application of formal methods in many cases. The application of formal methods just to the safety-critical parts of a system is a valid approach, but it requires that the safety-critical parts be identified and delimited. No general technique for isolating the safety-critical components of systems is available, however. In addition, it is not clear how to determine the properties of a system (or part of one) that are relevant to its safety. Again, no technique so far has been widely accepted, and most applications of formal methods try to establish properties chosen in a non-rigorous manner. Though clearly useful, this utility is informal—such properties do not contribute formally to the overall *system* dependability analysis. In summary, though formal methods are of value, it is not clear how they should be applied nor how to use the fact that they have been applied in system dependability analysis.

Given this situation, an integrated approach that: (a) addresses both the software and hardware elements of a system; and (b) exploits the tremendous potential of formal methods is needed. In this paper, a comprehensive approach to system dependability analysis based on traditional techniques for risk analysis is summarized. The approach models software as a set of interacting components based on the structure of the software. By viewing software this way, software analysis can be integrated fully into the models used presently for hardware. The resulting composite models provide details of those conditions in which hazards might occur as a result of erroneous software operation thereby identifying precisely where attention to software dependability must be focused. As such, these conditions can be the target of formal analysis so that confidence is gained about the right properties of the right parts of a software system.

3. A Component Model of Software Dependability

Both simple life testing and reliability growth models ignore the structure of software when obtaining estimates of software failure rates thereby requiring either that impossibly large numbers of tests be performed or that failures induced in one component by another be ignored. Unfortunately, however, if one appeals to formal methods as an alternative approach, one is faced with the fact that formal methods do not provide stochastic estimates and so cannot be used easily in place of testing. And, as we have already noted, it is not possible to identify precisely where or how such methods can be applied effectively to just parts of large systems.

Traditional dependability analysis techniques, such as fault-tree analysis and failure-modes-and-effects analysis, are performed for hardware-only systems using complete knowledge of the internal design. Typically, the software in software-based systems cannot be analyzed this way because the interactions between components have either been ignored or not obtained

rigorously. Clearly, software components such as functions and tasks interact extensively, but this is not to be the case (or is assumed not to be the case) in archetypal hardware-only systems.

Since techniques that model software as a monolithic entity have not achieved sufficient fidelity, we have developed an approach in which software is modeled as a graph with components as nodes and interactions as edges. An event associated with the failure of a software component then appears as a separate entity in the system fault tree. However, traditional *quantitative* analysis cannot be undertaken without further analysis because of the component interactions. *Qualitative* analysis, however, is possible and the comprehensive system fault tree allows those parts of the software whose failure might lead to a hazard to be identified easily.

In our approach, interactions are determined based on a component-interaction model and then minimized using architectural techniques. The resulting fault tree is then analyzed quantitatively using extensions to fault tree analysis that include dependencies [5].

Of critical importance is the fact that the failures of individual *software* components now appear in the *system* fault tree. This permits system design decisions to be taken to reduce vulnerabilities, but, more importantly, it indicates what aspects of the software will benefit most from the use of formal methods and how. For example, if a software component is deemed to be critical because the fault tree shows that its failure would lead to a hazard with unacceptable probability, then the component can be subjected to detailed formal analysis. If it can be shown to be correct via proof, then its probability of failure can be assumed to be close to zero and increased confidence gained in the system's safety. The role of formal methods is then clear.

4. Component Interaction Model

In developing an analysis-by-components approach to modeling software, the first step is to determine how one software component can affect another. There are, of course, a multitude of ways that this can occur, but there is no basis in either models of computation or programming language semantics for performing a comprehensive analysis.

We chose to approach this problem by viewing component interaction as a *hazard* and basing our analysis on a fault tree for this hazard. In this way, we have documented, albeit informally but rigorously, all possible sources of software component interaction. The fault tree is quite large and we cannot include it here in detail. The events in the fault tree are based on the semantics of a typical procedural programming language, and the results apply to all common implementation languages such as Fortran and C.

In order to reflect the syntactic structure of procedural languages accurately, we define the term component to mean either (a) a function in the sense of a function in C, (b) a class in the sense of a class in C++, or (c) a process in the sense of a task in Ada. We make no assumptions about how components can provide services to each other (in a client/server model) or collaborate with each other (in a concurrent model) or otherwise interact.

As an example of the interaction model, figure 1 shows the top of the component-interaction fault tree. With no loss of generality, in this fault tree we consider only two components because there can be no interaction between components if there is no pair-wise interaction. Since information flow between A and B is symmetric, only one of the cases need be considered.

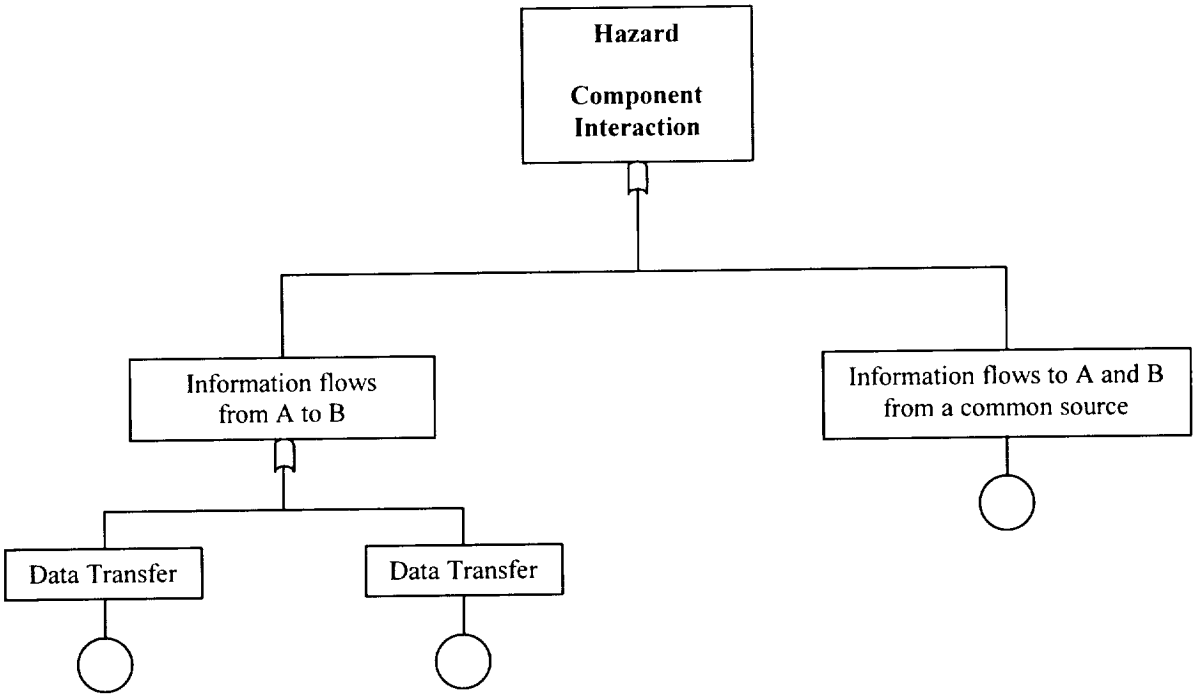


Figure 1: Fragment of component interaction fault tree

In the first level of the partial tree shown in figure 1, component interaction can be caused by information flow from A to B or by a common source of information. Thus, these are the two events shown. Note that component interaction does not necessarily mean *intended communication* in any format. Rather, it includes both intended and non-intended interaction between components. In addition, information flow does not mean just transfer of data. Flow of control is also information in the sense of the analysis that we wish to perform.

The complete interaction model derives sources of interaction in all semantic areas including shared data, memory management (e.g., one task consuming all memory thereby causing others to fail), task communications (e.g., priority inversion and deadlock), and exception generation and propagation.

5. Design Techniques for Analyzability

If analysis using our software component model is to be complete, it is essential that interactions between components that have to be analyzed always be detectable. Analysis of the component interaction model indicates that several potential causes for unwanted interaction cannot be discovered by static analysis of the system. Dynamic scheduling of functions and dynamic resource allocation, for example, have the potential for leading to failure under circumstances that are unpredictable. Similarly, other characteristics of software designs have the potential for increasing the complexity of the analysis or even making it infeasible.

Analytic feasibility requires that these sources of interaction be eliminated and this requires that certain restrictions be imposed. Both imposing the restriction and showing that a system meets them is best achieved by explicit use of design choices, for example:

- All resources must be statically allocated.
- All scheduling actions must be static.
- Execution times of components must be bounded.
- Inter-task communications must be synchronous.

This list, although not exhaustive, illustrates the properties that were derived from the component interaction model. Provided the complete set of design restrictions is met, all interactions between components of a software system can be analyzed. Achieving analytic feasibility of complex software systems using architectural techniques such as these is not unique to the approach we have developed. The SAFEBus architecture [4], for example, used in the Boeing 777 air transport enforces several of these properties to facilitate the safety analysis of the final system.

6. Quantitative Analysis

The final step in the approach that we have developed is quantitative analysis of complete systems including both hardware and software. The composite fault tree contains nodes describing failure events of all system components and all interactions between components are known. To complete the part of the quantitative analysis associated with the software nodes, we have developed an extension to the cut-set technique employed with conventional fault trees. The extension, termed *hazard-causing sequences,* involves enumerating all sequences of software component failures that could cause a hazard and analyzing each such sequence to show that its probability of occurrence is sufficiently small. If this analysis reveals a sequence whose probability of occurrence is not sufficiently small, formal techniques (perhaps combined with certain restricted forms of testing) can be applied to the sequence in order to either reduce the probability to a sufficiently small value or to show how the system design can be modified to make the associated sequence less critical.

7. Summary

In order to better model the dependability of complex software-based systems, we have developed an approach that uses the design of the software (viewed as a set of interacting components) as a basis for analysis. This approach permits the critical elements of the software to be identified and subjected to analysis using formal techniques. The approach, therefore, permits a clear determination to be made of the most appropriate application of formal methods to a large system and permits the results of formal analysis to be included in comprehensive system dependability analysis.

At the workshop we will describe the approach in detail, present the complete component interaction model, discuss the analytic techniques used in analysis of the composite fault-tree model, and illustrate the approach using analysis performed on parts of the design of an experimental nuclear reactor control system.

References

1. Brocklehurst, S.; Littlewood, B. *Techniques for prediction analysis and recalibration.* Chapter 4. In: Lyu, M. R., (ed). **Handbook of Software Reliability Engineering.** IEEE Computer Society, Los Alamitos, CA, 1995.
2. Butler, R. W.; Finelli, G. B. *The infeasibility of quantifying the reliability of life-critical real-time software.* In: **IEEE Transactions on Software Engineering**, v. 19, n. 1, p. 3–12, Jan. 1991.
3. Craigen, D; Gerhart S; and Ralston, T. *An international survey of industrial applications of formal methods.* National Institute of Standards Technology, U.S. Department of Commerce, (March 1993)
4. Hoyme, K.; Driscoll, K. *SAFEbus.* In: **Proceedings of the 1992 IEEE/AIAA 11th**. Digital Avionics Systems Conference, Seattle, WA, USA, 5Ð8 Oct. 1992, p. 610, 68–73. IEEE, New York, NY, USA, 1992.
5. Pullum, L. L.; Dugan, J. B. *Fault tree models for the analysis of complex computer-based systems.* In: **Annual Reliability and Maintainability Symposium. 1996 Proceedings. The International Symposium on Product Quality and Integrity**, Las Vegas, NV, USA, 22Ð25 Jan. 1996, p. 200–7, 1996.

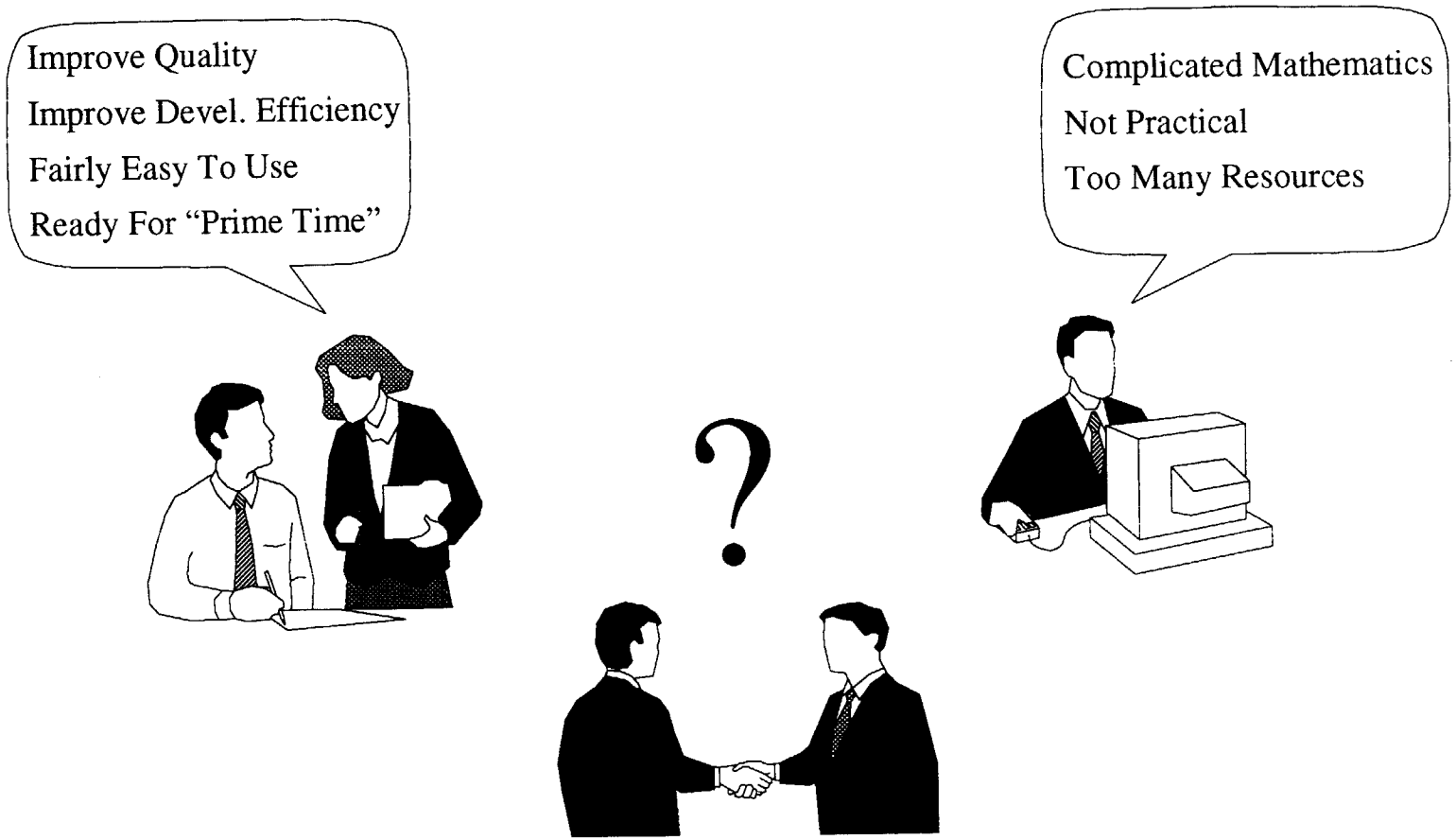
INTEGRATING FORMAL METHODS INTO SOFTWARE DEPENDABILITY ANALYSIS

John C. Knight and Luís G. Nakano

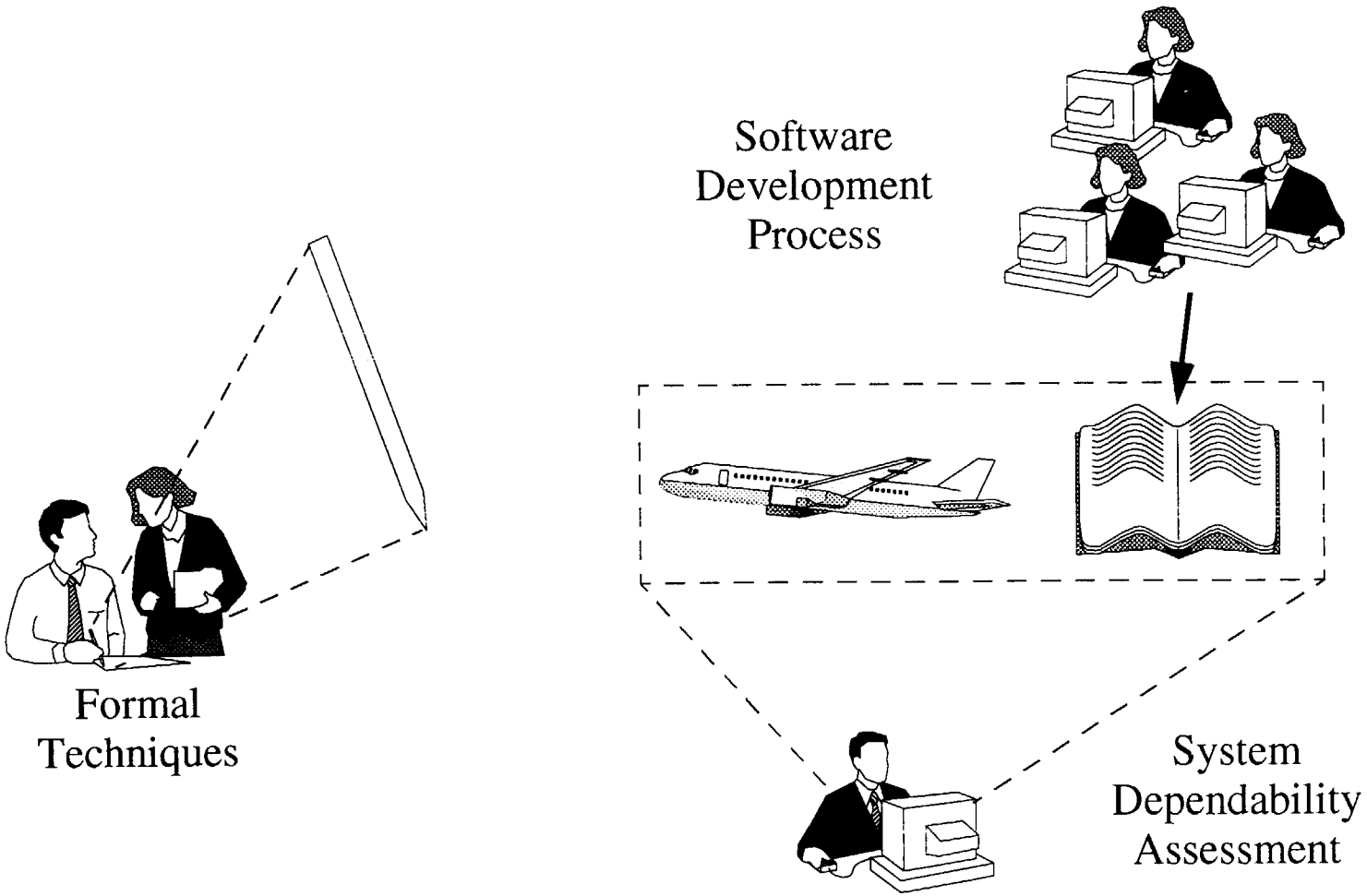
Department of Computer Science
University of Virginia
Charlottesville
Virginia, USA

UVA
Department of Computer Science

THE USE OF FORMAL METHODS



HYPOTHESES


Software
Development
Process
Formal
Techniques
System
Dependability
Assessment

QUESTION

- There Are Many Excellent Formal Techniques, Including:
  - Formal Specification
  - Specification Analysis
  - Refinement/Reification
  - Correctness Proofs
  - Property Proofs
- But, Comprehensive Formal Analysis Of Large Systems Is Impractical, So:

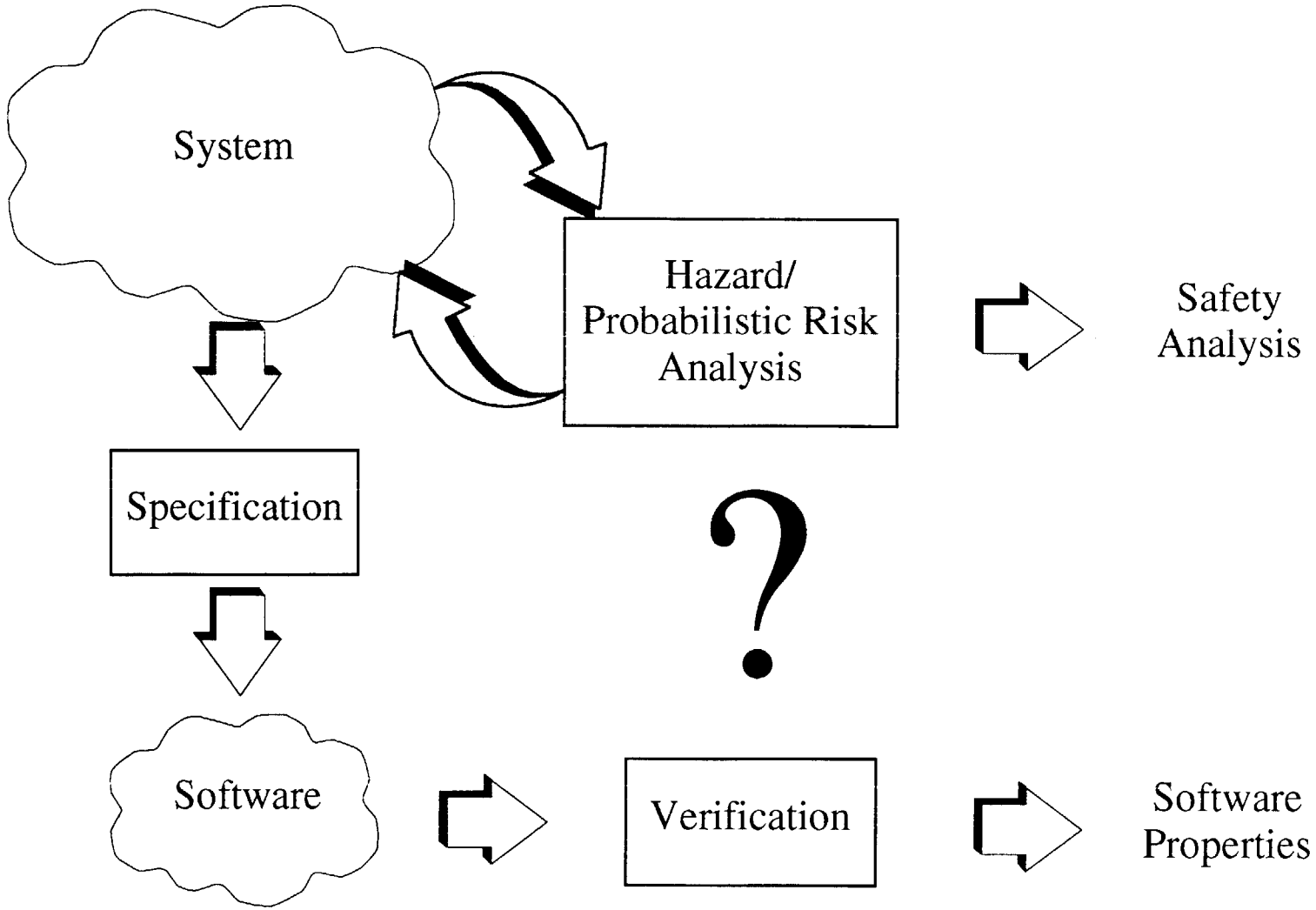
To which parts of a system should formal techniques be applied?

- Need A Means Of Determining Where They Can Be Applied Most Effectively

Application And Evaluation

- Formal Specification Has Been Used Extensively, For Example:
  - Statecharts — By Airbus, Guidant, Boeing
  - Z — By IBM, Praxis
  - SCR/Core — By NRL, Lockheed
  - PVS — By JPL, Rockwell Collins
- Various Evaluations Performed, For Example:
  - Craigen, Gerhart, Ralston (NIST)
  - Ardis et al. (Lucent)
- Previous Work Did Not Address Breadth Of Use:
  - Evaluation Criteria Tended To Be Technical, Performance Oriented

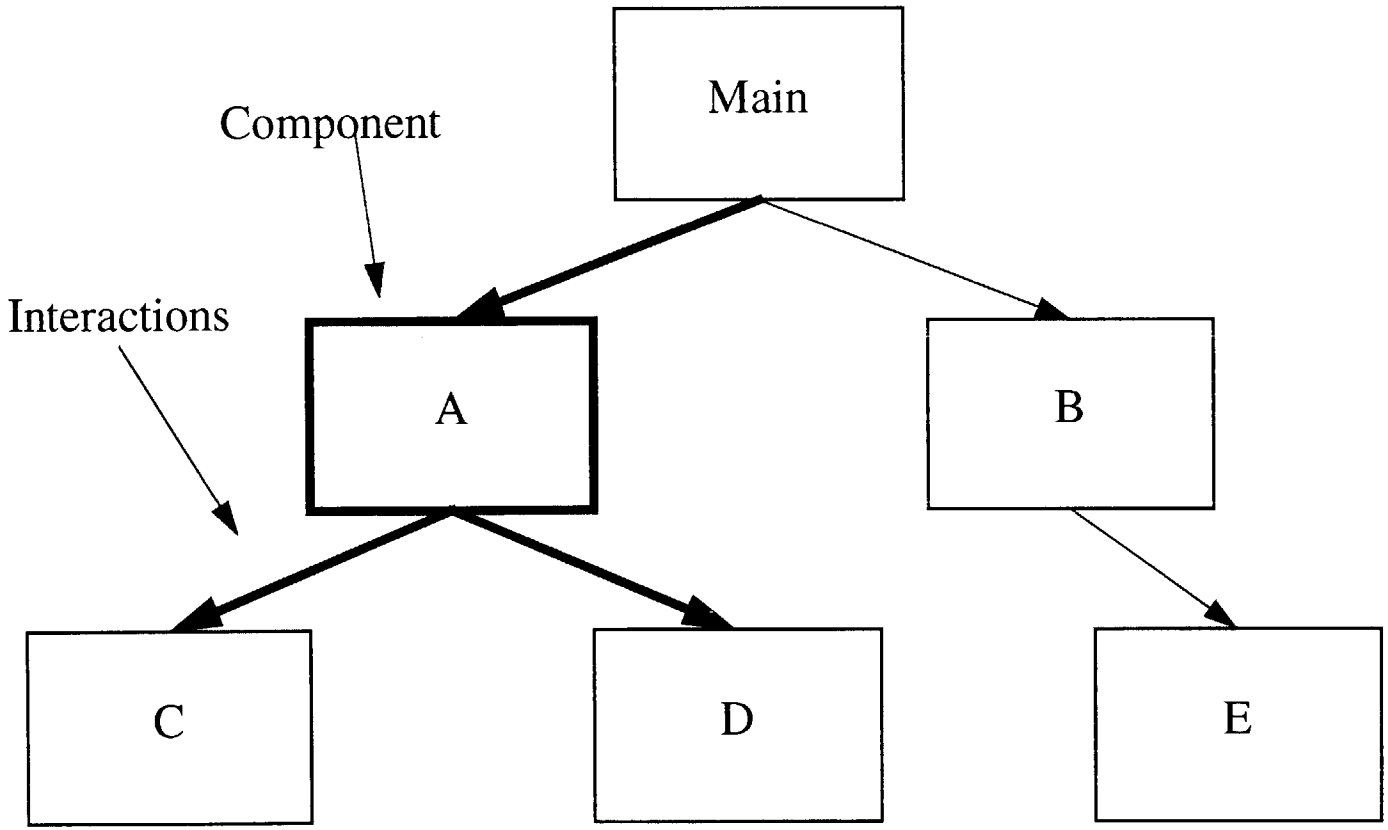
WHY BE FORMAL IN SW DEVELOPMENT?



SW IN SYSTEM DEPENDABILITY ASSESSMENT

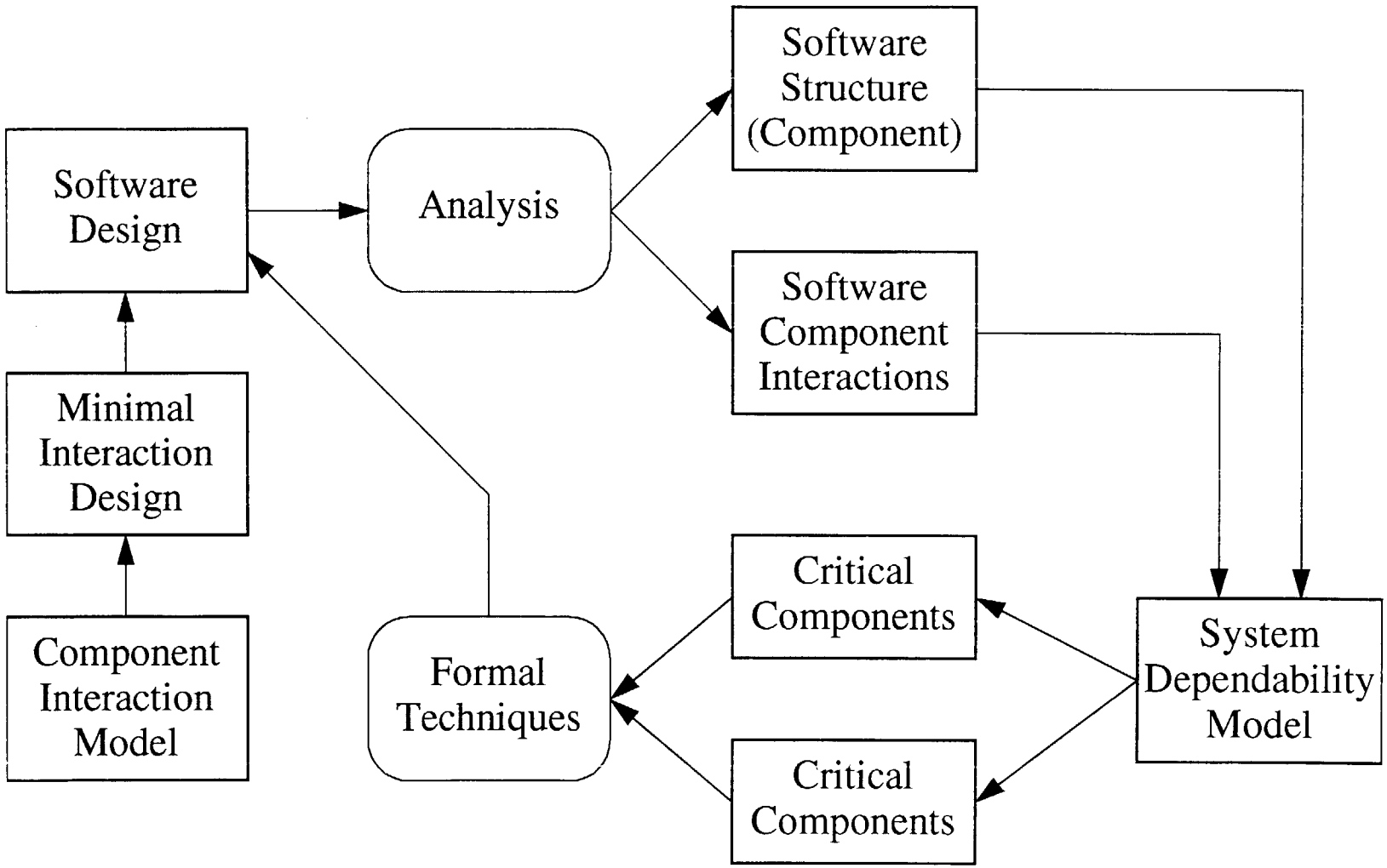
- Software Provides Lots Of Functions:
  - Are They All Critical?
  - Do They All Fail The Same Way?
- Typical Practices—Assume Only One Software Failure Event, And:
  - Try To Measure Probability Of Failure By Life Testing
  - Or Set Probability Of Failure To One
  - Or Maybe Zero
  - Or Maybe Model Using A "Reasonable" Distribution
- Software System Life Testing:
  - Its Generally Infeasible (Butler And Finelli)
  - Its Worse Than That (Ammann, Brilliant, And Knight)

SOFTWARE STRUCTURE



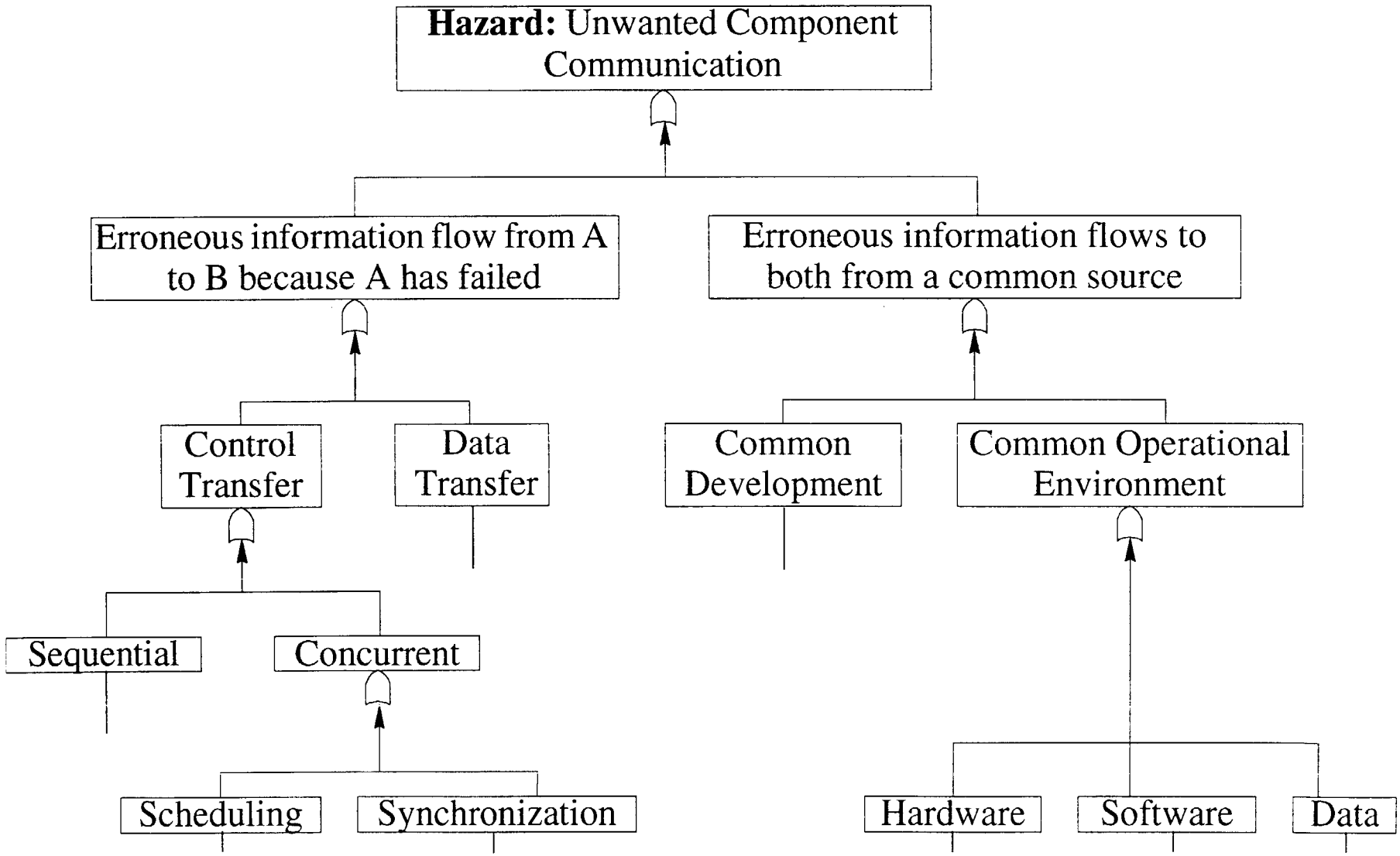
- Model Software Using Its Component Structure
- Analogy With Hardware Analysis

COMPONENT MODEL OF SW DEPENDABILITY



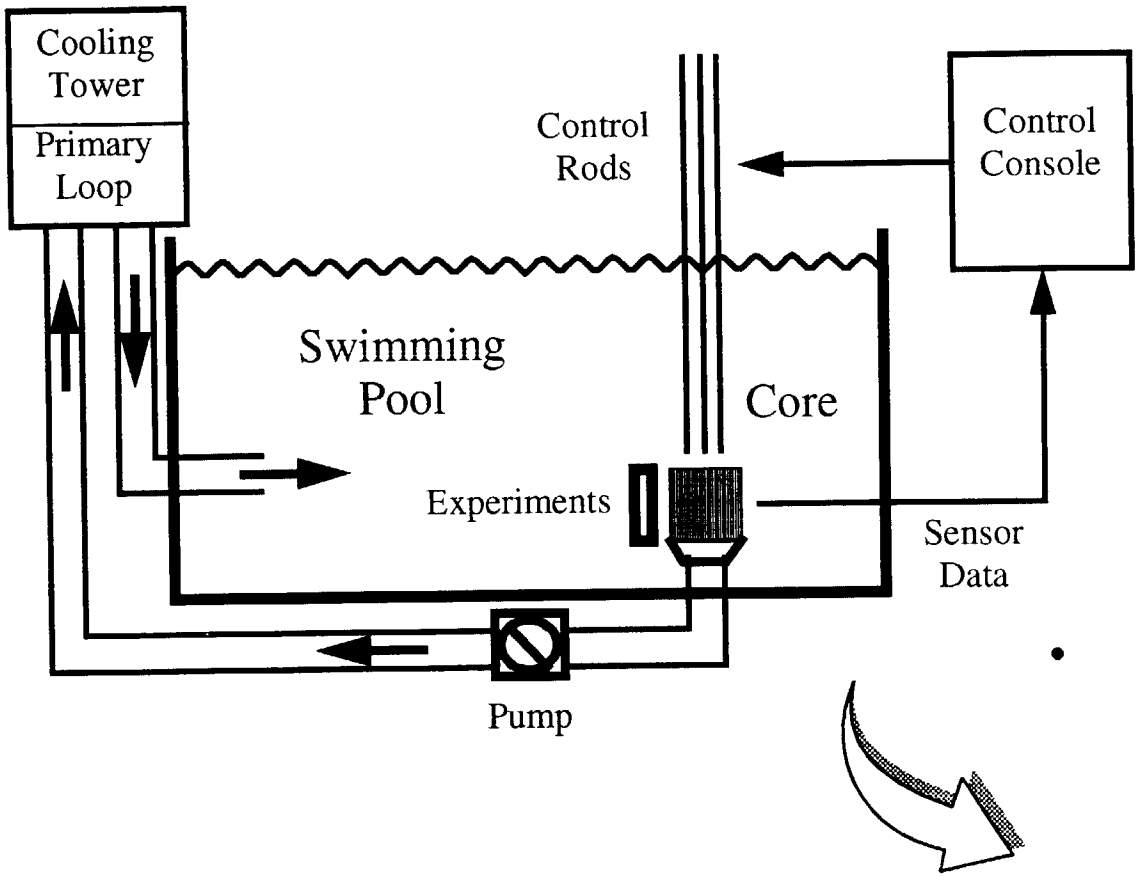
UVA
Department of Computer Science

COMPONENT INTERACTION MODEL



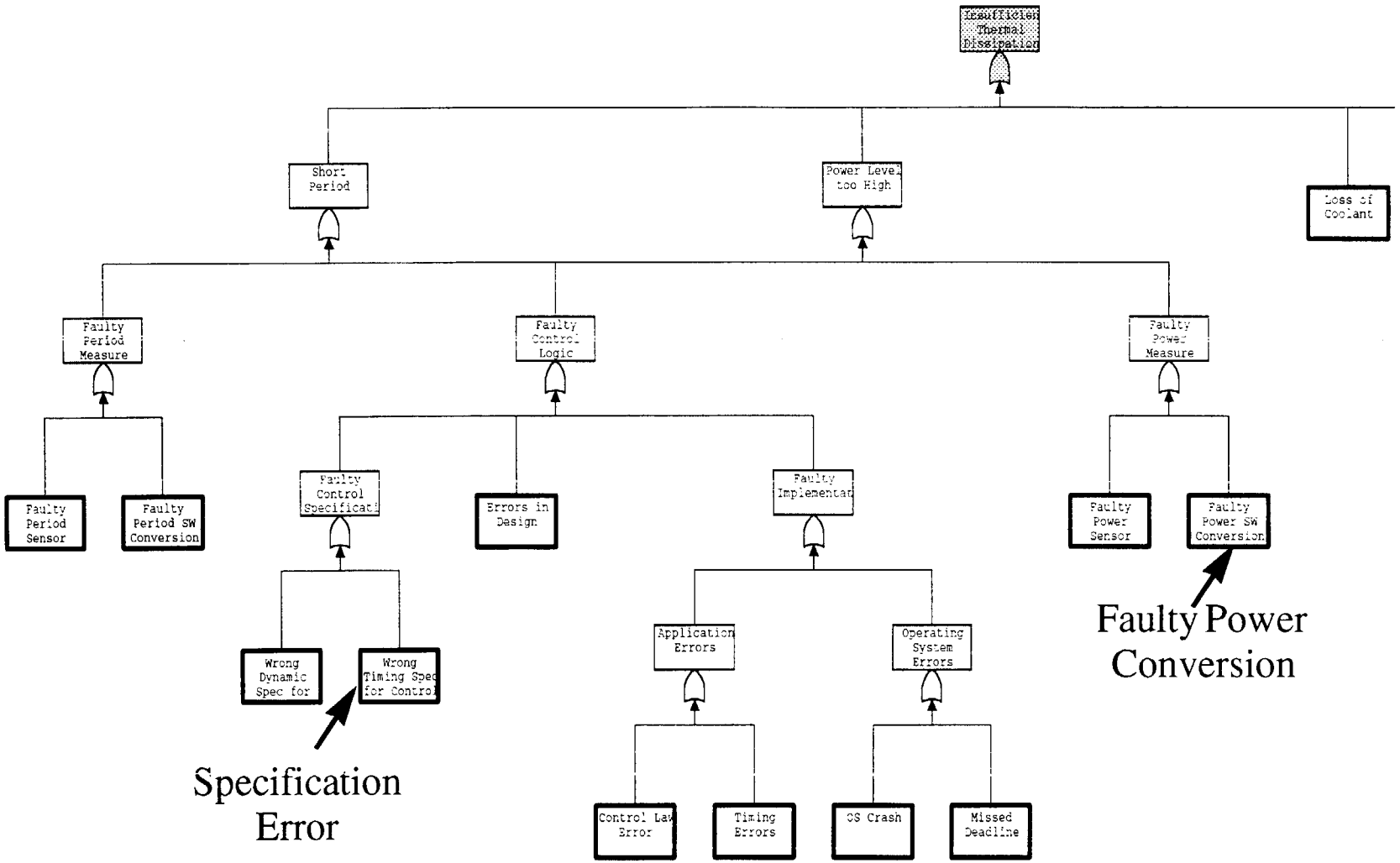
UVA
Department of Computer Science

UNIVERSITY OF VIRGINIA REACTOR SYSTEM



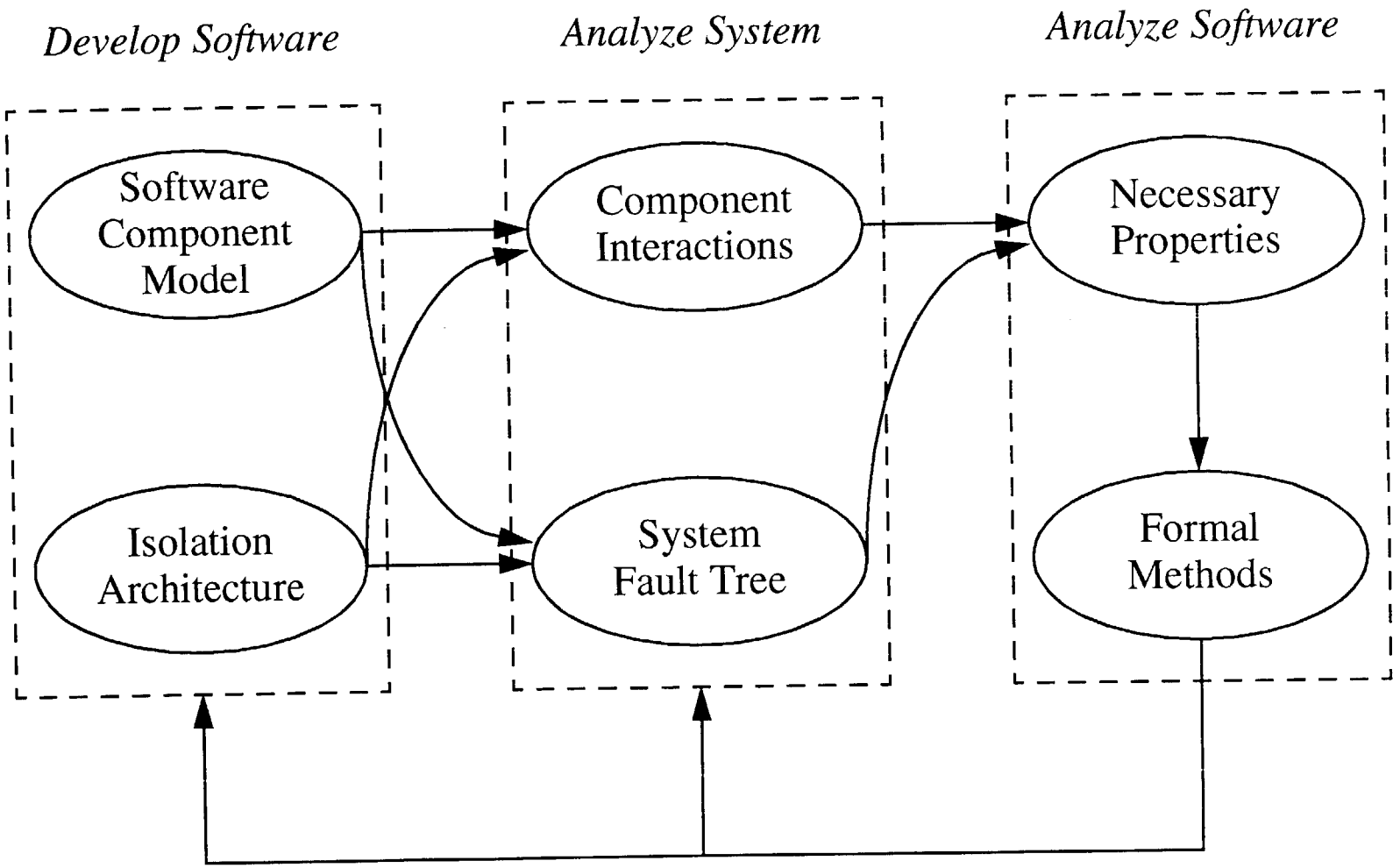
- *Lots* Of Software:
  - Operating Systems
  - Network
  - Window System
  - User Interface

SOFTWARE FAULTS



UVA
Department of Computer Science

INTEGRATED ROLE OF FORMAL TECHNIQUES



CONCLUSIONS

- Software Not Well-Integrated With System Dependability Assessment
- No Precise Role For Formal Techniques In Development
- Formal Techniques Don't Contribute Directly To System Dependability Assessment
- Component Model Of Software Dependability Developed
- Based On:
  - Component Software Design
  - Component Interaction Model
  - Integration Of Component Analysis In System Fault Tree
  - Enhancement Of Software Design Based On Fault-Tree Analysis

(12)
IN-61

AN ADAPTIVE SOFTWARE RELIABILITY PREDICTION APPROACH

*Meng-Lai Yin** *Lawrence E. James* *Samuel Keene* *Rafael R. Arellano* *Jon Peterson*
Raytheon Systems Company
Loc. FU. Bldg. 675, M/S AA341
1801 Hughes Drive, Fullerton, CA 92834 USA

ABSTRACT

Software reliability analysis is inevitable for modern systems, since a large amount of system functionality is now dependent on software, and software does contribute to system failures. Although extensive research efforts have been devoted to the field of software reliability, there is no single consensus model available. On the other hand, most software reliability models are based on software failure data collected from the project. This creates a problem for the designers since, during the early stage, software failure data are not available. This paper presents the approach we took to deal with the above issues. The adaptive approach presented here continuously adjusts and evaluates the performance of the models as the software development proceeds. For the early-stage prediction, a simple and straightforward method is introduced which can be used when no failure data are available. This process, which is based on the adaptive approach and includes the early-stage prediction method, has been implemented in a software intensive development program in progress.

INTRODUCTION

As more and more failures attributed to software are observed, it is recognized that software reliability analysis is an inevitable task. However, although several software reliability models have been proposed [6], there is still no "standard" model. In reality, the needs of software reliability prediction force people to choose one (or more) models so that some software reliability numbers can be provided. The problem with this approach is that, at the beginning of a system development, there is no failure data available. Thus, no one knows which model best describes the software product. This approach is referred to as the blind approach.

Another approach is to apply various models and the results are compared with actual failure data at the end of the project. This way, the performance of different models can be evaluated [12]. The problem is, the software reliability can not be estimated until the very-late stage of the development, when software is almost ready to be delivered. This approach is referred to as the autopsy approach.

To cope with the above problems, we propose an approach that analyzes software reliability *adaptively*. That is, software reliability is modeled as the software development proceeds. First, we provide a rough estimation, to start the whole process. As the software is being developed, failure data become available, and software reliability can be predicted progressively. Comparing the actual failure data with the predicted numbers, we can see the trend of the software failure behavior, and determine which models are the most appropriate ones. When the software development reaches the final stage, modeling experience is also becoming more mature. The ultimate goal is to provide software reliability estimation using the model that best characterizes the failure behavior of the particular software product. Not only that, this process continuously provides estimation at each phase of the system development based on the most current failure information.

Note that even at the beginning of software development where failure data are not available, some assurance that the design is meeting its requirements is desirable. Therefore, a method that can provide a reasonable estimation before any actual failure data available is a benefit to the program. In a

* Contact author. Email: mlyin@west.raytheon.com. Tel: 714-446-4269. Fax: 714-446-3137.

survey provided in [6], three models have been identified as the "early-phase' models, i.e., Gaffney and Davis' phase-based model [3], Agresti and Evanco's Ada software defects model [1], and the Air Force's Rome Lab model [9].

The basic philosophy of these early-phase models is to do a prediction *based on as much information as possible.* For example, the phase-based model requires the information of discovered faults found during the design and implementation phases [3]; the Rome Lab's model considers a very comprehensive list of factors [9]. The Ada software defects model requires 4 product and 2 process characteristics [1]. Although detailed information is desirable, they are not necessarily available, or they may be very costly to obtain at the early stage of the program. In this paper, we propose a cost-efficient method, called the *early-stage prediction*, to be added to the adaptive prediction process for software reliability.

This adaptive process with the early-stage prediction method has been implemented in a software development program in progress. As more experience is gained and more failure data are collected, the performance of the early-phase prediction method is improved.

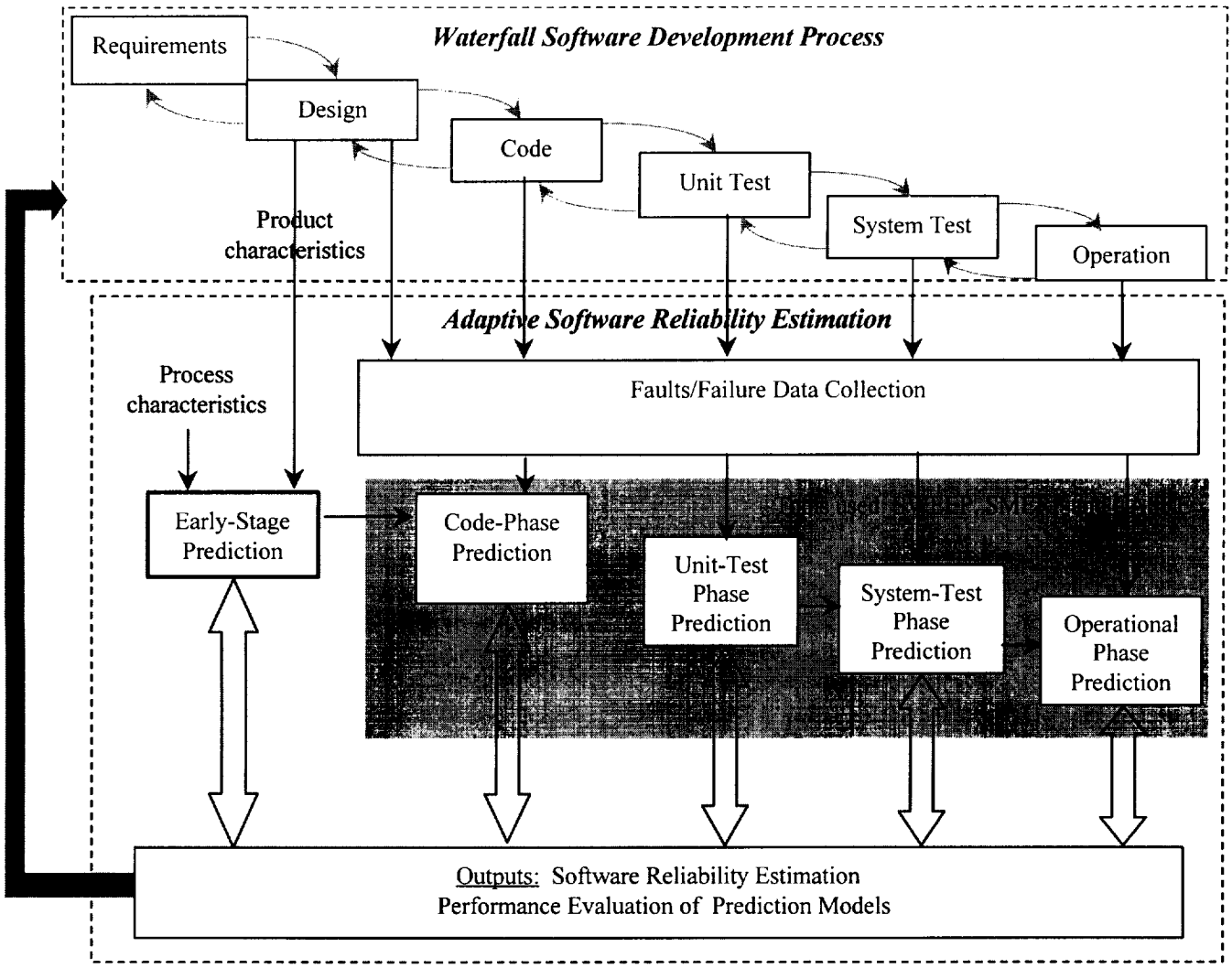


Figure 1. The Adaptive Software Reliability Prediction Process

THE ADAPTIVE APPROACH

The Process

The adaptive approach is integrated into the software development process, as shown in Figure 1. The

waterfall-software-development process [6] is used as the basis. As shown in the figure, a software product starts with some set of requirements, followed by *design, code, unit test, system test,* and the *operation* phases.

Five prediction activities are identified, i.e., early-stage prediction, code-phase prediction, unit-test-phase prediction, system-test-phase prediction, and finally the operational phase prediction. The early-stage prediction will be described in detail later. Once the software has been designed and implemented, information about discovered faults can be obtained, and code-phase estimation can be performed. The unit-test and system-test phase predictions can be conducted once those test data are available. When the software reaches the field (operational phase), software reliability growth is projected over its future use[1]. As failure data are being collected, the performance of the models can be evaluated. The outputs are not only the predicted software reliability number, but also an evaluation of the models. As illustrated in Figure 1, the outputs are fed back into the estimation process so that the software reliability models can be refined and justified. Moreover, these outputs are fed back into the development process to improve the product.

Tools Consideration

When faults/failure data are available, tools such as SWEEP (SoftWare Error Estimation Program), SMERF (Statistical Modeling and Estimation of Reliability Functions) and CASRE (Computer-Aided Software Reliability Estimation) can be applied. In particular, our process uses CASRE for the operational phase prediction and SMERF for the code-phase, unit-test-phase and system-test-phase prediction. SMERF and CASRE utilize the same set of models. SMERF is developed at the Naval Surface Warfare Center (NSWC) [6], and CASRE is developed in 1993 at Jet Propulsion Lab[10]. Eleven models are supported, i.e., geometric model, Jelinski/Moranda De-Eutrophication model, Littlewood and Verrall's Bayesian model, John Musa's basic execution time model, John Musa's logarithmic poisson model, Non-homogeneous Poisson (execution time), Brooks and Motley's discrete model, generalized Poisson model, Non-homogeneous Poisson (interval data), Scheiderwind's Max Likelihood model, and Yamada's S-shaped growth mode [6].

SWEEP is an implementation of the phase-based model [3]. It makes use of fault statistics obtained during the technical review of requirements, design, and the coding to predict the reliability during test and operation. Thus, SWEEP can be used before testing (after coding). On the other hand, CASRE and SMERF can be used in the system test phase. None of the above tools can be used for the very early-stage prediction where no fault or failure data are available. A methodology that provides estimation for this situation is the topic of the next section.

EARLY-STAGE PREDICTION

The purpose of this method is to provide a rough estimation on various software reliability measurements, based on the limited information. In particular, the only information required are the size of the software, measured by source lines of codes (SLOC), the maturity of the development process[2], and the schedule. Since only a rough estimation is expected, accuracy is not a main concern for the early-stage prediction. Instead, accuracy is the goal of the overall adaptive process, which will be achieved by continuously refining various models. The two basic assumptions are (1) the time between software failures is exponentially distributed (2) the occurrence of a failure is followed by the removal of the corresponding fault[3].

There are many research efforts devoted to the topic of imperfect software debugging. In particular, the asymptotic properties of software failure rates have

[1] The issues of asymptotic properties of software reliability have been studied [11], and different methods have been proposed.

[2] The software development process level, such as the SEI(Software Engineering Institute) CMM(Capability Meturity Model) or the ISO 9000 series of standard by the International Organization for Standardization, have been proposed to assist the assessment of inherent faults [5].

[3] This implies that we assume there is a one-to-one mapping between the faults and failures.

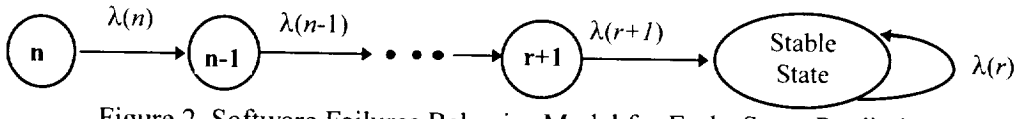


Figure 2. Software Failures Behavior Model for Early-Stage Prediction

been studied [11]. There is always a possibility that new faults will be introduced when removing a software bug. However, from a statistical point of view, the number of newly introduced faults is less significant when the total number of remaining faults is (relatively) large. It is only when the software product is reaching the mature stage, where the number of remaining faults and the number of introduced faults are in the same order of magnitude, should the imperfect debugging be concerned. This phenomenon is captured in our model as the "stable" state. Figure 2 shows the model that describes the behavior of software failures.

In this model, a software program is estimated to have n inherent faults at the beginning of the estimation. An assumption is made that the corresponding fault is removed when a software failure occurs. This will bring the software to the next state where the number of faults is decreased (one at a time). This process continues until the software reaches the stable state. In the stable state, the asymptotic failure rate phenomenon is observed. A failure rate function $\lambda(i)$ is used in this model. This failure rate function $\lambda(i)$ can be described in many different ways, according to the software failure behavior. For example, it can be described as a linear increasing function that is in proportion to the number of remaining faults, i.e., $\lambda(i)=i\lambda$. Or, the failure rates can be described as a logarithmic increasing function, i.e., $\lambda(i)=ln[i]\times\lambda$. This failure rate function should be a function of λ. The value of λ is then calculated based on the model, the parameters, and the failure rate function specified.

The key of this method is to find out the value of λ, using the information of the size of the code, the software process maturity level, and the duration T. T is the duration from the beginning time the software is measured (t_0) to the time the software is 'stable' (t_d). Theoretically, the selection of t_0 can be any time, for example, the time the software is finished compilation or the beginning of various phases indicated in the waterfall process.

According to [2] and [5], the actual failure data from different programs show that the stable time is approximately 4 years after delivery for a new program release. The stabilization period might be reduced to two years for subsequent program releases.

Once the starting time and the "stable" time are determined[4], the next step is to estimate the number of inherent faults, denoted as n, and the number of remaining faults, denoted as r.

Estimating the numbers of inherent faults and remaining faults

In order to solve the model described in Figure 2, the number of inherent faults, i.e., n, and the number of remaining faults, i.e., r, need to be determined. The inherent faults are the faults existing at time t_0; remaining faults are the faults existing at time t_d. A wide-used method to determine the number of inherent faults is through the use of fault density[5].

There are several studies on estimating the fault density. Musa's survey [7][8] provides fault density estimated for different software life-cycle phases. As presented in [8], the mean inherent fault density remaining at the beginning of different phases is estimated based on actual failure data from many different programs. As an example, the inherent fault densities for different phases are summarized in the following table.

Table 1

Phase	Faults/KSLOC
Coding (after compilation/assembly)	99.5
Unit Test	19.7
System Test	6.01
Operation	1.48

(copied from [8], Table 5.4)

[4] Note that this is only a rough estimation.

[5] Although Hatton [4] disagrees with this approach, the size of the code times the fault density is commonly used in the field.

The work of Aagresti and Evanco's Ada software defects estimation method [1] recognizes the differences in the way organizations develop software for software reliability prediction. Both *process* characteristics and *product* characteristics are considered in the overall software defects model. Moreover, Keene [5] proposed an approach that applies the software process levels and the size of the code to predict the number of inherent faults. As an *example*, the following table shows the relationship of the inherent fault densities at the beginning of the operational phase and the software process levels.

Table 2

SEI CMM Level	Faults/KSLOC
5	0.5
4	1.0
3	2.0
2	3.0
1	5.0
Un-rated	6.0

For the value of r, i.e., the number of remaining faults, the observation in [2] and [5] suggests that, after four years of deployment, the number of software faults be reduced to a level less than 10% of the level at deployment. Thus, one way to conservatively estimate the value of r is to use 10% of the fault density estimated at the beginning of the operation phase.

Calculating λ

Define a sequence of non-negative real-valued infinite random variables X_0, X_1, X_2, ... , X_i, ... Each of these random variables represents the time between two consecutive failures. Recall that exponential distribution has been assumed. The value of λ is assessed by utilizing the relation that $E[X_1 + X_2 + \ldots + X_{n-r+1}] = E[X_1] + E[X_2] + \ldots + E[X_{n-r+1}]$. In other words, we have the equation $1/\lambda(n) + 1/\lambda(n-1) + \ldots + 1/\lambda(r+1) = T$[6]. Given the failure rate function $\lambda(i)$ and the values of n, r and T, the value of λ can be calculated.

Stable State MTBF

Plugging λ into the failure rate function with parameter r, i.e., $\lambda(r)$, the stable state MTBF can be estimated, i.e., $1/\lambda(r)$.

Expected Number of Failures Occurred

The expected number of software failures occurred by time t is calculated in the following way. First, we estimate the state the software is expected to be in at time t. This can be done by calculating $\Sigma(i \times P_i)$, where i is the number of existing faults (the state number), and P_i is the probability that the software is in state i at time t. Denote the expected number of existing faults at time t as k. Then, the expected number of software failures that have occurred by time t is $n-k$.

Software MTBF prior to the Stable State

Suppose at time t, the number of existing faults is predicted (by the above method) to be k, then the MTBF at time t can be estimated as $1/\lambda(k)$. This can be interpreted as the mea time between software failures if no further faults are removed.

Software Reliability

According to the standard definition of reliability[7], the software program's reliability is the probability that at time t, the software is still in state n (no failure has occurred yet). This reliability number depends on the value of λ, which in turn is dependent on the other parameters and the failure rate function specified.

Improvement of the Method

Although this method only provides a rough estimation, the actual data collected in the later phases will give us feedback on the method and the parameters used. Furthermore, the experience from actually implementing the process will improve the overall approach, which can be used for other programs. For example, the fault density level at the stable state used now is 10% of the level when the software is deployed. This number can be refined or justified, as more experience is gained. The accumulation of this experience over time can be added to the confidence in the reliability parameters, which can then be used in upcoming programs.

In the next section, an example is used to demonstrate the early-stage prediction, and the feedback gained from later phases prediction activities.

[6] For exponential distribution $1-e^{-\lambda t}$, the expected value is $1/\lambda$.

[7] The reliability is defined as the probability that the component operates correctly throughout the interval $[t_0, t]$ given that it was operating correctly at time t_0.

AN EXAMPLE

In this example, we consider a software product whose size is 360 KSLOC (K Source Lines Of Code). The software process applied is rated as SEI CMM Level 4. The duration T is assumed to be 4 years. Based on this information, the early-stage prediction method suggests the fault density at the beginning of the operation phase is 1.0 per thousand lines of code, i.e., 360 faults, if the process-driven fault density model is applied (Table 2). If Table 1 is used, then the fault density at the beginning of operation phase is 1.48, i.e., 533 faults. These two models give us a rough estimation on the number of inherent faults at the beginning of the operation phase. According to the discussion presented in the previous section, we can derive various software reliability measurements based on this information.

As the software development is progressing, faults/failures data are collected. The tool SWEEP was used to perform a phase-based model prediction [3]. Eight phases were specified, e.g., preliminary design, detailed design, code, unit test, integration test, final test, system test, and operation phases. Figure 3 through Figure 7 show the adaptive predicted fault density for each phase based on different sets of available failure data. Specifically, Figure 3 is the prediction made at the beginning of code phase, when only the defects found in preliminary design and detailed design are known. Figure 4 is the prediction made at the beginning of unit test, and so on.

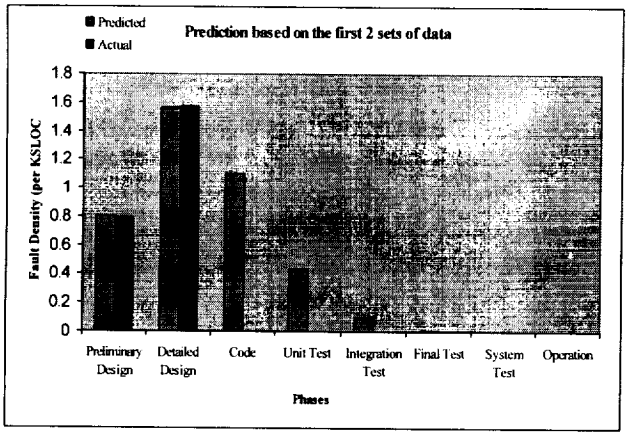


Figure 3. SWEEP Prediction based on 2 sets of data

The stage of the project is currently at the beginning of system test. Therefore, only the failure data up to the final-test phase are available. The predictions show that the operation phase fault density based on the most updated failure data, i.e., 0.99, is very close to our early-stage predictions (1.0 if using process-driven model, 1.48 if using Musa's survey). This example demonstrates that we can earn more confidence in the model that we chose at the earlier stage, by the predictions performed at the later phases.

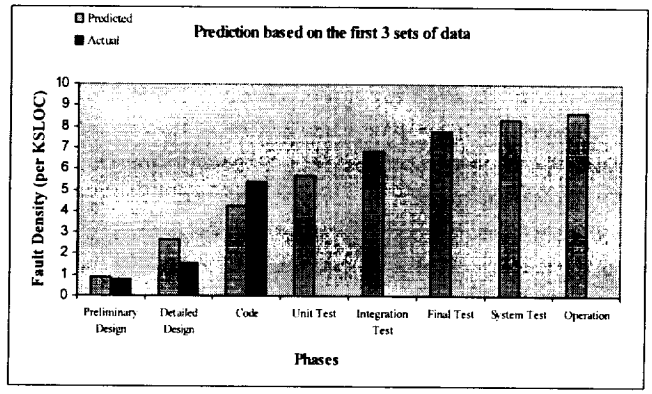


Figure 4. SWEEP Prediction based on 3 sets of data

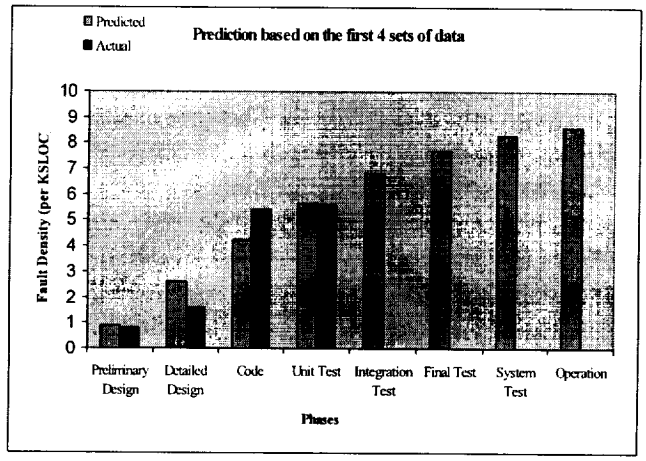


Figure 5. SWEEP Prediction based on 4 sets of data

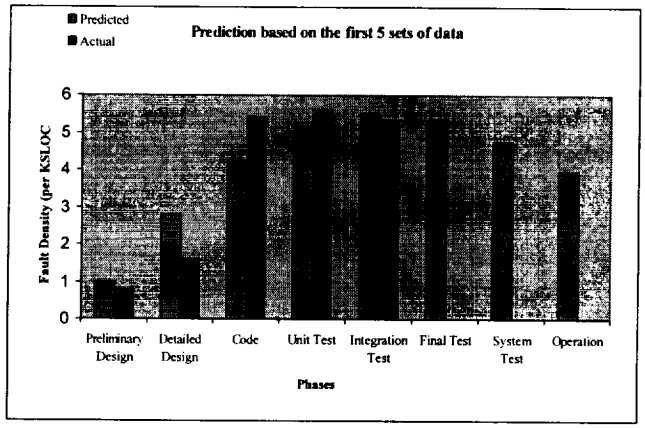


Figure 6. SWEEP Prediction based on 5 sets of data

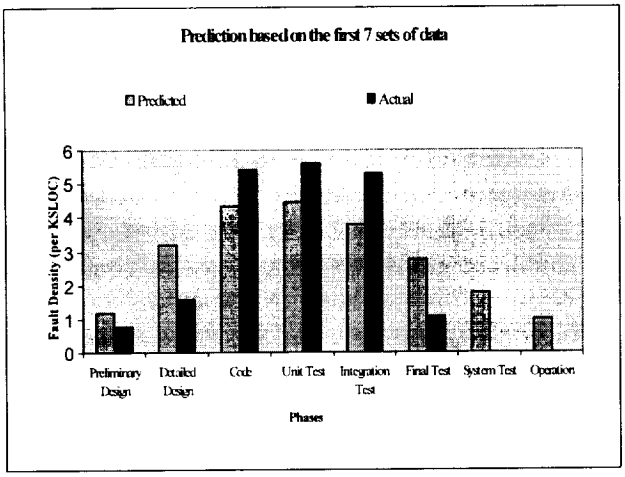


Figure 7. SWEEP Prediction based on 6 sets of data

CONCLUSION

We have presented an adaptive approach, which is integrated with the software development process, to estimate the software failure behavior. This approach has been implemented in an ongoing software development program. The key feature of this method is that the prediction is improving as the software proceeds. Our basic philosophy is that, since the software product is evolving continuously, the software reliability prediction should be improving continuously.

Moreover, a method that can assess software reliability in the early stage is presented. This method requires only very limited information about the software product and the process. The asymptotic property of software failure rates is recognized in the model. While most early-phase software reliability prediction methods focus on how to provide a precise prediction with the limited information, we provide a rough estimation as a starting point of the overall prediction process. The accuracy of the estimation is the goal of the overall process. The approach presented here is readily performed and should provide adequate initial software reliability estimation. As more experience in this early-stage prediction is gained, the method can be improved and benefit other software development products.

REFERENCES

[1] W.W. Agreti, and W.M. Evanco, "Projecting Software Defects From Analyzing Ada Design," IEEE Transactions on Software Engineering, Vol.18, No.11, Nov.1992, page 988-997.

[2] Ram Chillarege, Shriram Biyani, Jeanette Rosenthal, "Measurement of Failure Rate in Widely Distributed Software," *Fault Tolerant Computing Symposium (FTCS),* 1995, page 424-433.

[3] J.E. Gaffney and Davis, C.F., "An Automated Model for Software Early Error Prediction (SWEEP)," Proceedings of the 13th Minnowbrook Workshop on Software Reliability, July 1990.

[4] L.Hatton, "Reexamining the Fault Density – Component Size Connection," IEEE Software, March 1997, pp. 89-97.

[5] S.J. Keene, "Modeling Software R&M Characteristics," ASQC Reliability Review, Part I and II, Vol 17, No.2&3, 1997 June, pp.13-22.

[6] Michael R. Lyu (editor), *Handbook of Software Reliability Engineering*, McGraw-Hill, 1996.

[7] John Musa, "A Theory of Software Reliability and Its Application," IEEE Transactions on Software Engineering, Vol. SE-1, No.3, Sep. 1975, page 312-327.

[8] John D. Musa, Anthony Iannino, Kazuhira Okumoto, *Software Reliability - Professional Edition,* McGraw-Hill, 1990.

[9] Rome Laboratory (RL), Methodology for Software Reliability Prediction and Assessment, Technical Report RL-TR-92-52, volumes 1 and 2, 1992.

[10] A.P. Nikora, "CASRE User's Guide," Jet Proopulsion Laboratories, August 1993.

[11] M.C.J. Van Pul, *Statistical Analysis of Software Reliability Models*, Stichting Mathematisch Centrum, Amsterdam, 1993.

[12] A. Wood, "Predicting Software Reliability," IEEE Computer, Nov. 1996, pp. 69-77.

Raytheon

An Adaptive Software Reliability Prediction Approach

Meng-Lai Yin * Lawrence E. James Samuel Keene
Rafael R. Arellano Jon Peterson

Software Reliability Prediction Approaches

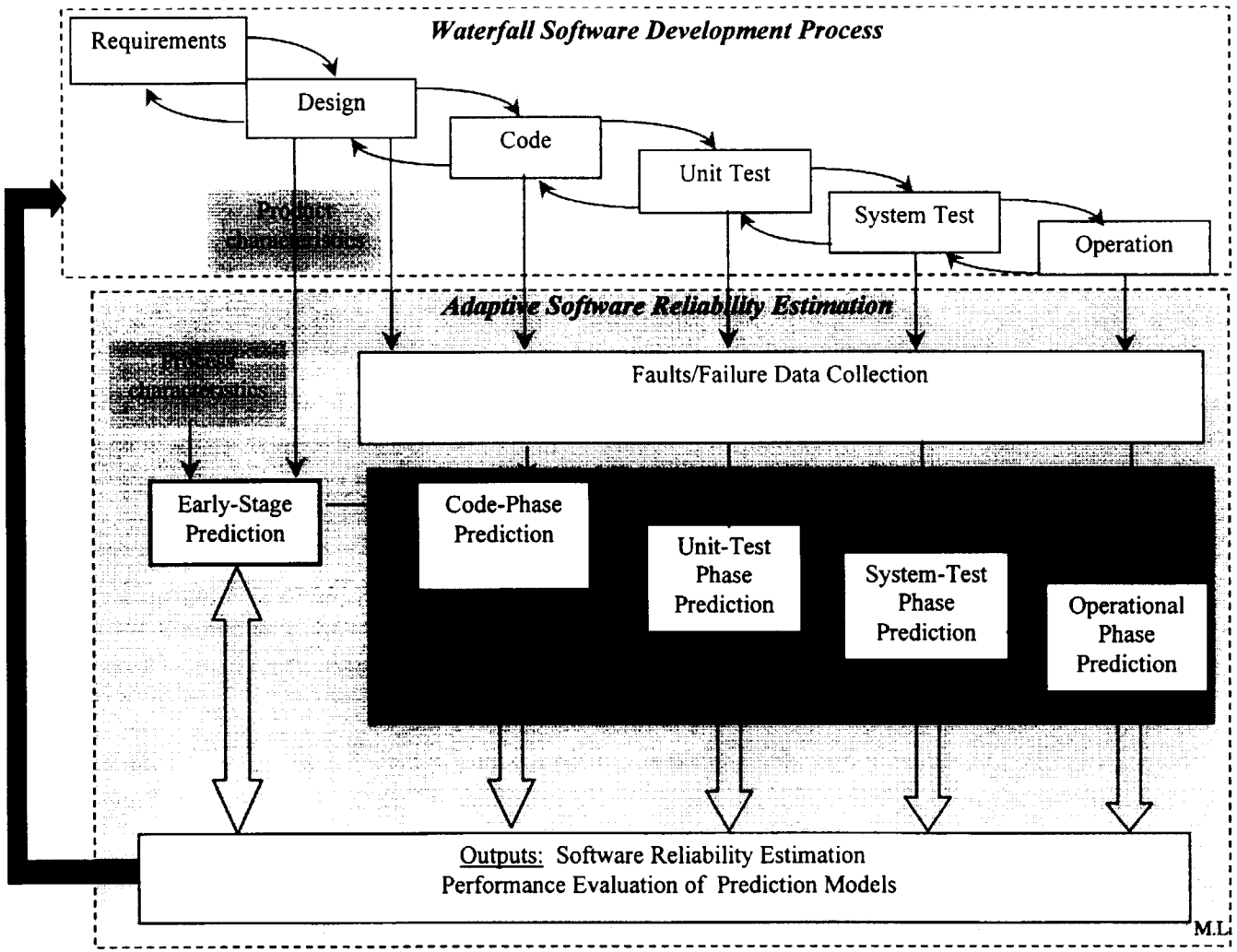
- No "standard" model to be used
- Blind approach
- Autopsy approach
- Adaptive approach
  - analyze software reliability adaptively and progressively

Adaptive Approach for SW Rel. Prediction

- Software reliability is modeled as the software development proceeds
- The prediction can be refined and justified progressively
- Continuously provide estimation based on the most current failure information
- Prediction at the beginning of software development, when no failure data are available, is also necessary (early-stage prediction)
- This process has been implemented in a software development program in progress

Raytheon

The Process



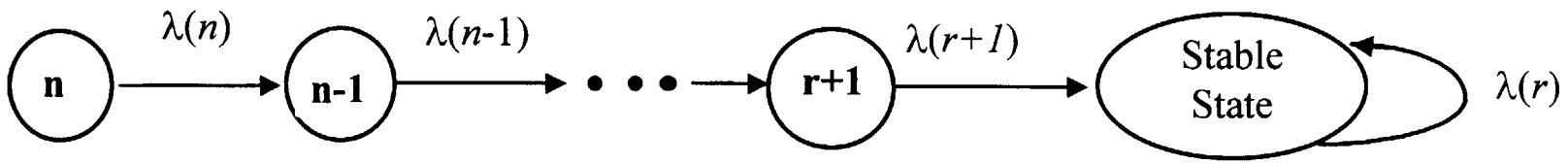
Tools Consideration

- SWEEP (SoftWare Error Estimation Program)
  - phase-based model
- SMERF (Statistical Modeling and Estimation of Reliability Functions)
  - various growth models
- CASRE (Computer-Aided Software Reliability Estimation)
  - various growth models

Early-Stage Prediction

- Provide rough estimation on various software reliability measurements, based on limited information (Accuracy is the goal of the overall adaptive process, not the main concern of the early-stage prediction)
- Information needed are
  - the size of the code
  - the software process maturity level
  - the schedule of software development

The Early-Stage Prediction Model



n: the number of inherent faults

r: the number of remaining faults

Stable State: the state where the software failure rate remain the same (asymptotic property)

$\lambda(i)$: failure rate function

Calculation Process

1. Determine n, r, and T
   - n = size of code * inherent fault density
   - r = n * remaining faults percentage
   - T is from the beginning of operational phase to the time the software reaches stable state
2. Specify failure rate function
3. Solve λ from $T = 1/\lambda(n) + 1/\lambda(n-1) + \ldots + 1/\lambda(r+1)$
4. Various measurements can be obtained from the result

Example 1. Fault Densities for different Software life-cycle Phases

Phase	Faults/ KSLOC
Coding	99.5
Unit Test	19.7
System Test	6.01
Operation	1.48

(copied from Musa[8], Table 5.4)

Raytheon

Example 2. Fault Densities for different SEI CMM Level

SEI CMM Level	Faults/ KSLOC (at the beginning of operation phase)
5	0.5
4	1.0
3	2.0
2	3.0
1	5.0
Un-rated	6.0

(copied from keene[5])

Example

- 360 KSLOC
- SEI CMM Level 4 Process
- 4 years to reach the stable state after software deployment
- 1.48 faults/KSLOC (533 faults) for phase-based inherent fault density prediction
- 1.0 faults/KSLOC (360 faults) for process-maturity-level-based prediction

Summary

- The adaptive approach and the early-stage prediction method have been implemented in an on-going software development program and provided adequate prediction
- As more experience is gained, the process and the method will be improved, and will benefit other software development products.

Session 5: Verification & Validation

Model Checking Verification and Validation at JPL and the NASA Fairmont IV&V Facility
F. Schneider, Jet Propulsion Laboratory, S. Easterbrook, NASA IV&V Facility,
J. Callahan and T. Montgomery, West Virginia University

Using Model Checking to Validate AI Planner Domain Models
J. Penix, C. Pecheur, and K. Havelund, NASA Ames Research Center

V&V of a Spacecraft's Autonomous Planner through Extended Automation
M. Feather and B. Smith, Jet Propulsion Laboratory

Performing Verification and Validation in Reuse-Based Software Engineering
E. Addy, NASA/WVU Software Research Laboratory

Model Checking Verification and Validation at JPL and the NASA Fairmont IV&V Facility[1] -

Frank Schneider, Jet Propulsion Laboratory, California Institute of Technology, Steve Easterbrook, NASA IV&V Facility, Jack Callahan and Todd Montgomery, West Virginia University
Contact: Francis.L.Schneider@jpl.nasa.gov

13
IN-61

Abstract

We show how a technology transfer effort was carried out. The successful use of model checking on a pilot JPL flight project demonstrates the usefulness and the efficacy of the approach. The pilot project was used to model a complex spacecraft controller. Software design and implementation validation were carried out successfully. To suggest future applications we also show how the implementation validation step can be automated. The effort was followed by the formal introduction of the modeling technique as a part of the JPL Quality Assurance process.

Introduction

Following the pilot use of model checking at NASA JPL and the IV&V Facility [1], and at NASA Ames[2], we have followed five steps in introducing model checking to the Quality Engineering process at JPL. First, references [1] and [2] show model checking to be an effective tool in validating the behavior of spacecraft systems. Second, our model checking results were then carried forward to validate the software implementation for the presence of design anomalies. Third, having validated the implementation by hand, we show how the process can be automated. Fourth, we have documented the process to be used in a development environment by incorporating and generalizing the above elements. Finally, we are engaged in applying the methodology developed here on future spacecraft.

Model Checking as a Validation Tool

We use model checking to mean the process of (1) abstracting a partial specification from requirements and design elements for a reactive system and (2) applying reachability analysis to the resulting partial specification to validate that it has properties of interest. A reactive system is one that takes input from its environment at unpredictable times and responds according to a specific set of rules. We have previously shown model checking to be an effective tool in validating the behavior of a fault tolerant embedded spacecraft controller [1]. That case study shows that by judiciously abstracting away extraneous complexity, the state space of the model could be exhaustively searched allowing critical functional requirements to be validated down to the design level. The system we validated was a two-fold redundant spacecraft controller. It consists of a prime system that controls the spacecraft bus and a backup system. The backup system receives synchronization information from the prime system via the spacecraft bus. The purpose of the system is two fold. First, it has to respond to and repair must-fix-spacecraft faults.

[1] The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Second, it must complete execution of high priority sequences. To realize these goals two mechanisms were utilized.

First, the system uses a checkpointing scheme that allows:

- Execution to be frozen when a fault occurs
- Repair of the fault somewhere in the spacecraft
- Rollback to the start of the last incomplete subsequence
- Resumption of sequence execution

Accordingly, the checkpointing scheme allows efficient sequence execution since completed subsequences need not and in many cases can not be repeated. The checkpointing scheme requires three seconds of aging for each new checkpoint before the new checkpoint is considered to have been encountered. This is caused by fault leakage detection time such that a fault at the end of a previous subtask may not be detected until up to three seconds after the beginning of a new subtask. This could mean that the fault precluded instructions at the end of the previous subtask from being executed.

Second, the overall redundancy of the system made up of prime and backup controllers allows the entire prime controller to fail. Failure is detected by the backup system that then becomes prime and takes over execution where the failed system halted. The backup system becomes prime; takes over control of the spacecraft bus; completes repairing the fault; rolls back to the start of the last incomplete subsequence and resumes execution of the sequence. Figure 1 illustrates the architecture involved. Further details can be found in reference [1].

The initial abstracted design state space contained about 2^{87} states. By this statement we mean that we estimate there to be 2^{87} different combinations of variable values and conditions that completely describe every possible configuration of the spacecraft controller. There are five types of faults such that the controller is required to respond to one type of fault at a time. Because fault detection and recovery requirements could be handled one-at-a time, the requirements were partitioned into five equivalence classes accordingly reducing the state space to be searched significantly. The state space was further reduced by removing states from the finite state machine representation that did not contribute to the checkpointing scheme we were attempting to validate. This gave rise to a new estimate of about 100, 000 states. The resulting Harel Chart [3] for the abstracted spacecraft controller is that shown in Figure 1.

Example: Sequence execution segment:

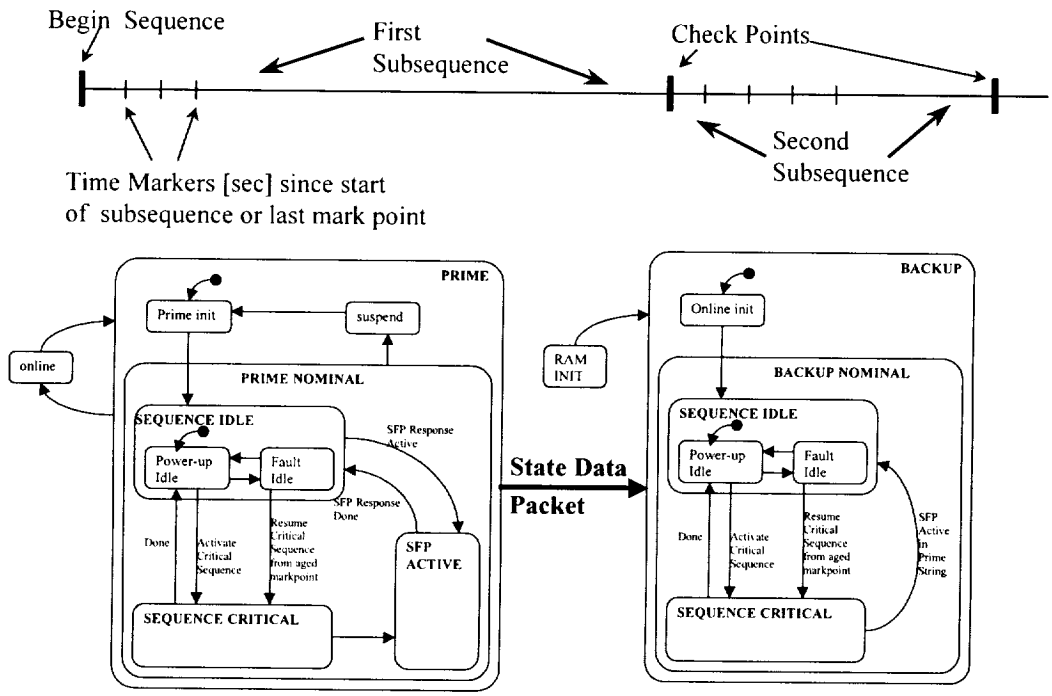


Figure 1

The validation was accomplished with the SPIN model checking system [4]. Six separate rollback requirements on the rollback scheme were validated. Three anomalies were uncovered with the model checker traversing about 130,000 states for each anomaly with run time being approximately 30 seconds for each anomaly.

Anomaly one resulted from repeated prime failure causing loss of synchronization with the backup system. This result occurs when the prime system experiences repeated intermittent failures possibly due to the same fault, and such that the prime system repairs the fault in less than one second. According to our model this would mean that notice of the fault would never be propagated to the backup system. Consequently, the backup system could get significantly ahead of the prime system in the execution of its own copy of the sequence. Then should the prime system subsequently fail, the backup system could roll back to an incorrect location. This anomaly is due to the ordering of processing described in the requirements specification.

Anomaly two depends on how faults are handled at the end of the sequence. Should a fault occurrence be detected up to within three seconds of execution of the last instruction, there would be no rollback after repair of the fault. This is the case since the last instruction in the sequence was not identified as a checkpoint. However, should a fault occur prior to the end of the sequence, according to the fault leakage detection rule there is no guarantee that all

instructions at the end of the sequence would have been successfully executed. Our validation run failed because our model assumed that once the sequence completed, the backup and the prime systems returned to the Power Up Idle state; accordingly, there would be no sequence to return to once the fault was corrected. This anomaly is due to a missing requirement.

The third anomaly concerns the occurrence of a fault 2 seconds after a checkpoint is encountered in the prime string. The prime string freezes its aging function at n + 2 seconds. Since faults that occurred in the previous second are not broadcast to the backup system until the current second it will continue to execute, aging its checkpoint by one further second. At this point the backup system receives notice of the fault and freezes its aging process. However, it now has an erroneous rollback point. Should the prime system subsequently fail, the backup system would roll back to an incorrect address. This requirement is an error in the detailed requirements. This is so since the error would not go away by making the checkpoint-aging buffer shallower or deeper. It would just make the anomaly occur at a different location.

Software Implementation Validation

We have subsequently validated the implementation for the presence of the three design anomalies. For this purpose we used a special purpose spacecraft simulator called the High Speed Simulator (HSS) [5, 6]. The simulator uses code identical to the real spacecraft. However, it is de-coupled from hardware and telemetry. Accordingly, its use as a test vehicle (1) is an accurate measure of system functionality and (2) it allows rapid turnaround on test suite creation, execution, and reporting of results.

The simulator allows test engineers to write test sequences for execution on the simulator. Given the data structures present in the spacecraft controller, a Tool command language (Tcl) program is written that orchestrates (1) the execution of the test sequence, (2) the extraction and printing of values of selected data attributes (3) the extraction and printing of any relevant time stamps and (4) fault injection scenarios and their responses.

1.1 Procedural Steps

We wanted to know if the software implementation contained the same anomalies as were found in the design. To determine this, we supplied the High Speed Simulator with a simple sequence program for execution. By injecting faults into the running sequence, the same problematic conditions would be set up in the implementation that were discovered by design validation. Our earlier validation work derived the design anomalies from a three-step process. First, the prime system would stop running freezing its check point ager in response to a fault occurrence somewhere in the spacecraft. Second, the prime system would load and begin execution of a fault recovery program. Finally, during its execution of the fault recovery program, the prime system itself would fail. To affect this same scenario in the software implementation, the prime system was commanded to do a cold boot at execution points in the implementation identical to those that caused the anomalies in the design validation. An operational backup system considers the prime system cold boot to be a prime system failure. It reacts by becoming prime itself; taking control of

the spacecraft bus; rolling back to the relevant earlier check point address if necessary; and resuming execution of the sequence program. For example, the third anomaly found in the design validation process occurs when the prime system fails after encountering a fault scenario that freezes its check point at second two in the aging process. This results in the new prime system rolling back to an inappropriate address due to a timing problem in the design. Accordingly, cold booting the prime system when it has aged its checkpoint by two seconds has the same effect as the two step process considered in the design case.

Detection of the presence of design anomalies in the implementation was done by selecting data structures for output identical to those used in the design case. These output data values taken together at any execution cycle represent the state of the implementation at a particular point in time. As the implementation executes, this 'state vector' describes a finite state machine that represents the implementation. This finite state machine is an abstracted finite state machine since it doesn't include all variables, only the ones considered relevant to the current validation. If a corresponding design anomaly is itself present in the implementation, the implementations' abstracted state vector will go through an equivalent sequence to that found in the design validation done earlier. In this case the work proceeded by outputting each state vector for the executing implementation. The output list was then manually examined line by line to look for the presence of anomaly states.

The input sequence program that was incorporated into the HSS Tcl interface program to check for the presence of anomalies in the implementation is shown in Figure 2.

```
IP      Mnemonic
800     BEGIN
803     NOP
805     NOP
807     NOP
809     NOP
80b     NOP
80d     CHECKPOINT
80f     NOP
811     NOP
813     NOP
815     NOP
817     NOP
819     CHECKPOINT
81b     NOP
81d     NOP
81f     NOP
821     NOP
823     NOP
825     END
```

Figure 2 Sequence Validation Program

To keep the analysis as straight forward as possible, each instruction was executed on one-second boundaries. A HSS Tcl interface program was written to generate the output state vector sequence of the abstracted implementation state machine. Schematically, the overall process is shown in Figure 3.

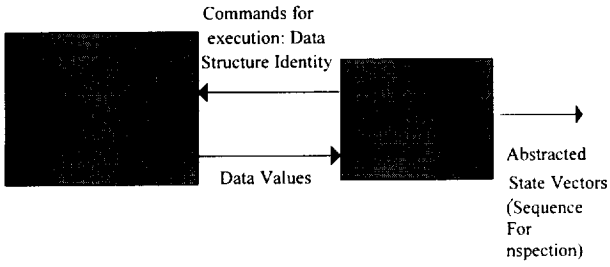


Figure 3: Implementation Abstracted State Machine

The implementation was validated at this point by simply looking at the results of the simulation by hand and recognizing that a design anomaly was or was not reproduced in the output. This means visually examining the output sequence labeled "Abstracted State Vectors" to check the rollback process functionality. Two of the three anomalies found in the design validation were present in the implementation. A brief summary of the results follows.

Implementation Anomaly Validation Results

The first anomaly resulted from repetitive errors that caused the prime and the backup system to get out of synchronization. Our design anomaly fault scenario required a series of prime-fault-repair sequences each of one-second duration or less. We did not see the first anomaly in the system. Further investigation with system engineers revealed that all faults take at least several minutes to repair. Therefore, repair time was extended so that anomaly one would not be seen.

The second anomaly occurs when a fault occurs less than three seconds after the sequence ends. In this case, there is no rollback. That is, once the sequence has been completed there is no rollback in response to an error injected inside the three-second-rollback window. Therefore, there is no guarantee that all instructions at the end of the sequence would have been carried out by the spacecraft. Accordingly, on this basis, the last instruction in the program should have been identified as a rollback point. Our technique demonstrated that the second anomaly was present in the implementation.

The third anomaly results from a fault that brings the prime system down when its aging buffer contains a check point rollback address that has been aged by two seconds. According to our model checking validation, this information would not get to the backup system until the following

second, thereby causing its two deep backup buffer to age its rollback address by an additional second. Consequently, its rollback address would be consistent with a three-second delay following a checkpoint when only two seconds had elapsed since the prime string had executed its last instruction. Prime system failure was again caused by cold booting the prime string at the point it had aged its checkpoint by two seconds. The subsequent rollback in the new prime system did not match the old prime's rollback address. Accordingly, our technique demonstrated that the third anomaly was present in the implementation.

The cold boot process is equivalent to the injection of a single fault that brings the prime system down. This process causes the overall spacecraft controller to fail to conform to requirements since control in the new prime system rolls back to an inappropriate location. Therefore, our technique also demonstrated that the overall system made up of prime and backup systems was not single fault tolerant.

All of these results were taken with respect to the spacecraft software as it existed on the High Speed Simulator.

Automating the Validation Process

Dillon and Ramakrishna show how test oracles can be generated from linear temporal logic specifications [7]. Log files generated from a running implementation can then drive these automata. The log files generated are used to drive requirements automata into accepting states should strings from the language they accept be traversed and output by the implementation. The automata are usually specified to check for requirements violations. Using these ideas, we have extended our work on design verification and validation [1] and applied it to the validation of the generic spacecraft controller's implementation. Our results used the output of the running spacecraft simulator system. The real time output was used to drive the automaton that represents one of the anomalies found by model checking. The resultant system was then made up of the spacecraft simulator; the test scenario generator, and automaton representing the requirement to be tested. The result system, called the Automated Validation System (AVS) did detect a counter example in the output indicating the presence of the design anomaly in the implementation. Additionally, the automaton has the capability to output the state vector trail taken by the implementation as it encountered the anomaly thereby giving information on how the anomaly develops as execution proceeds.

We have proposed that this concept be used as a fault protection mechanism on autonomous spacecraft. These spacecraft have self sufficient activities based on a set of high-level mission objectives carried on board the spacecraft. See for example [10]. The AVS would provide an effective and robust fault detection and response system for such spacecraft. The steps to be followed are outlined below.

1. Intercept the autonomously developed activity or action routine that the spacecraft is to carry out based upon and derived from the current mission profile.
2. Parse the mission profile or its more detailed on-board-generated requirements and derive from them the logical condition that represents the requirements that are to hold during and at termination of the executing action routine.
3. Express the logical condition in the linear temporal logic (LTL).

4. Define any macros that may be necessary to map the derived LTL automaton into any required ancillary form.
5. Produce the executable fault detection automaton from the LTL formula derived from steps 3 and 4.
6. Annotate the action routine so that it outputs an abstracted state vector representing an essential model of the action routine.
7. Couple the output from step 6 to the executable automaton produced in step 5.
8. Execute the overall action routine piping its real time output to the LTL automaton as it is produced. A fault condition will then drive the LTL automaton into one of its accepting states indicating that the associated requirement has been violated. When this condition is detected, respond autonomously to the fault. If this is not feasible, notify ground control and begin to safe the spacecraft as may be apropos of the situation.

1.2 Critique of Recommendations

The production of the executable LTL automaton cited in step 5 need not be a complex stumbling block. For example, safety conditions on total available power, maximum turn angles, antenna pointing and the like are easily quantified. The production of an automaton that checks for a liveness condition has already been illustrated in this section for design anomaly three of the spacecraft controller checkpoint process. Additionally, any of several other automata systems could be used depending on analyst choice.

The advantage of the system proposed here is that detailed knowledge of the underlying spacecraft software is not required nor would it ever be necessary.[2] Once the appropriate data structures governing the requirements are located, they can be output to the LTL automaton in the form of an abstracted state vector. Additionally, if the autonomous system makes use of an architecture analogous to the High Speed Simulator architecture, the process would be considerably more straightforward and precise. Here, once the appropriate data structures were identified, they would be easily tagged for inclusion in the abstracted state vector. All of this is again easily an automation step. In this latter case, consideration of the action routine per se would be minimized.

The usefulness of the procedure described here is that the automata theory is well understood, predictable, and easily programmable. Systems engineers would however have somewhat of a learning curve to become proficient in expressing requirements specifications in the linear temporal logic.

It might be argued that because the low level sequence and requirments are derived from high level mission objectives that AVS derived requirements are of course always going to work out. This would be the case should it be proven that the deductive logic used in producing detailed low level commands is valid in all circumstances. Therefore, one might argue what is the point of the procedure at all? However, such an argument would not be valid for it is a physical spacecraft in a physical universe dynamically making decisions about its environment. First, things can and do often go awry aboard the spacecraft. Single Event Upsets, bus failures, cameras jamming, valves freezing up or not opening properly on first try, squibs misfire and a host of others. Second, estimates of relative locations of external objectives can be misjudged. If distance estimates are accurate, angles and velocities can be way off due to low measurement resolution in the region where they are made.

Although the example in the text used a single AVS, the number of threads that might be running scales linearly. Accordingly, many such AVSs could be dynamically created as required and released when no longer necessary. Also, there is no reachability problem here as occurs in model checking due to the on-the-

[2] The Time Rover Company discusses their innovative test system at http://www.time-rover.com/SpecLang.html. It embeds LTL logic in the form of language statements within executable routines. Accordingly, it may be difficult to implement such a system within the context discussed here where on-the-fly validation is required for newly generated procedures.

fly-one-time nature of the problem here. We are in fact only interested in current behavior, not in all possible future behaviors. Therefore an AVS for maximum and minimum angles, power, closest distance of approach, and the like could be easily and dynamically configured for each scenario to be carried out.

Accordingly, the AVS system described here can provide a powerful, fast, reliable, first line of defense towards assuring mission success in the Laboratories' unmanned exploration efforts of the 21st century.

Formalization of the Model Checking Process

Having successfully shown the applicability of the model checking process on a spacecraft system, we decided to formalize the methodology. Our introduction of the modeling technique was via the use of a "process chart." A process chart is a flowchart that details the methodology that is applied to accomplish in our case a quality assurance technique. There are currently 21 processes for which process charts exist. Example process charts include Training, Risk Assessment, Requirements Assessment and Design Assessment. In addition to the process chart, each process also has an accompanying summary that details the contents of the process flow. The purpose of the process charts is to (a) document our own processes (b) to be able to convey in a clear and unambiguous way to our customers what our QA processes are and (c) in cases where customers want further assistance, we give guidance to them on how they can carry out their part of the resulting interface. The model checking verification and validation process chart includes high level guidance on how incremental design modeling is carried out over a project lifecycle for reactive systems and their components. This includes using the results of incremental design modeling to check to see if the design anomalies are present in an implementation. Callahan and Montgomery have discussed the use of this approach adopted here within the context of a model-checking environment in their development of the RMP Protocol [8]. In addition to using model checking to find design anomalies, the implementation is also checked in an incrementally evolving development environment. If a faulty design is subsequently corrected, the condition of the implementation can then be checked to see that it reflects the new design. Conversely, should the partial implementation get ahead of the partial design, then the implementation can be used to check the design. In this way an evolving partial implementation and a partial design can be driven to maintain phase coherence with each other. The process thereby yields an implementation that has a much higher confidence level associated with it. Additional details concerning this approach can be found in [9].

Ongoing Work

A test harness similar to the one we have used on the generic spacecraft validation effort is being constructed for a future series of deep space missions called X2000. Presently X2000 includes missions to Pluto and Europa. We are planning to use the validation methodology described here on the X2000 project.

Summary

We have shown model checking to be a viable and useful technology to apply towards making future spacecraft designs and their corresponding implementations more robust. A method was suggested whereby a validation scheme called the Automated Validation System could be used to provide an analytic framework to wrap conventional fault detection and response mechanisms

aboard autonomous spacecraft. The Verification and Validation process using model checking was formalized at the Laboratory by adding the Model Checking process to our Office 506 Quality Assurance process methodology system. Our effort at applying the technology to future spacecraft is a work in progress. We plan to publish a more detailed analysis of the results given in this paper.

References

[1] Francis L. Schneider, Steve M. Easterbrook, John R. Callahan, and Gerard J. Holzmann: Validating Requirements for Fault Tolerant Systems using Model Checking. ICRE 1998: 1-13.

[2] Michael R. Lowry, Klaus Havelund, John R. Penix: Verification and Validation of AI Systems that Control Deep-Space Spacecraft: ISMS 1997: 35-47

[3] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231-74, 1987.

[4] G. J. Holzmann, "The Model Checker Spin," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279-295, 1997.

[5] Reinholtz, WK and Robison, WJ.,III, "TheZIPSIM series of high-performance, high fidelity spacecraft simulators," *Proceedings AIAA/Utah State University Annual Conference on Small Satellites,* Aug 29-sept 1, 1994.

[6] Patel, K and Reinholtz, W and Robison, W, "High-speed simulator: A simulator for all seasons", *Proceedings International Symposium on Space Mission Operations and Ground Data Systems (SPACEOPS96,* Munich, Germany Sept 16-20 1996; pg 749-756

[7] L.K. Dillon and Y.S. Ramakrishna: Generating Oracles From Your Favorite Temporal Logic specifications: SIGSOFT'96 CA, USA 106-117

[8] J. R. Callahan and T. L. Montgomery: An Approach to Verification and Validation of a Reliable Multicasting Protocol: ISSTA 96: 187-194

[9] John Callahan, Francis Schneider, Steve Easterbrook: Automated Testing Using Model-Checking: Invited talk Bellcore division of Bell-Laboratories: 1996 - see also http://swayer/~sch/

[10] N. Muscettola, B. Smith, C. Fry, S. Chien, K. Rajan, G. Rabideau, and D. Yan. "On-Board Planning for New Millennium Deep Space One Autonomy," Proceedings of the IEEE Aerospace conference, Snowmass CO, 1997.

JPL

Model Checking Verification and Validation at JPL and the NASA Fairmont IV&V Facility -

Frank Schneider, Jet Propulsion Laboratory, Steve Easterbrook, NASA IV&V Facility, Jack Callahan, Todd Montgomery, West Virginia University

Funded by NASA's Software Program, Office of Safety and Mission Assurance UPN #232-08-5L

Model Checking: We use model checking to mean the process of (1) abstracting a partial specification from requirements and design elements for a reactive system and (2) applying reachability analysis to the resulting partial specification to validate that it has properties of interest. A reactive system is one that takes input from its environment at unpredictable times and responds according to a specific set of rules.

The Problem and its solution:

Goal: Show applicability for and to gain acceptance for use of model checking methodology in space craft development efforts in NASA spacecraft development projects

Response: In response to this goal a project manager made a design specification for a complex spacecraft (S/C)controller available to us

The Specification:

1. **Dual system with identical separate backup platform**
2. **Communication of state information from one to the other**

System - design purpose is two fold:

1. **Respond to and repair must-fix-external faults while**
2. **Completing high priority sequence execution**

Checkpointing scheme allows:

1. **Sequence execution to be frozen when fault occurs**
2. **Fault to be repaired**
3. **Rollback to start of last incomplete subsequence**
4. **Resumption of sequence execution.**

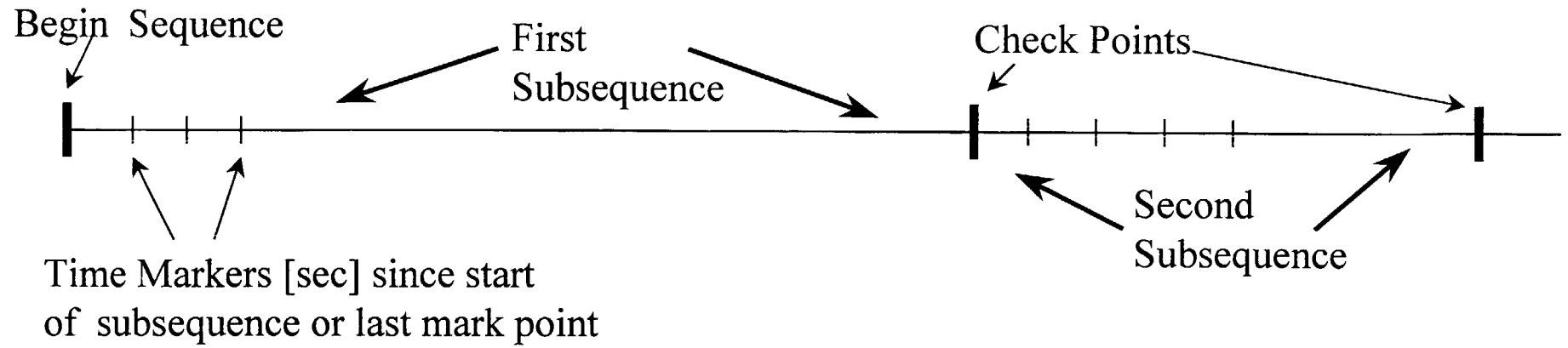
Checkpointing scheme requires

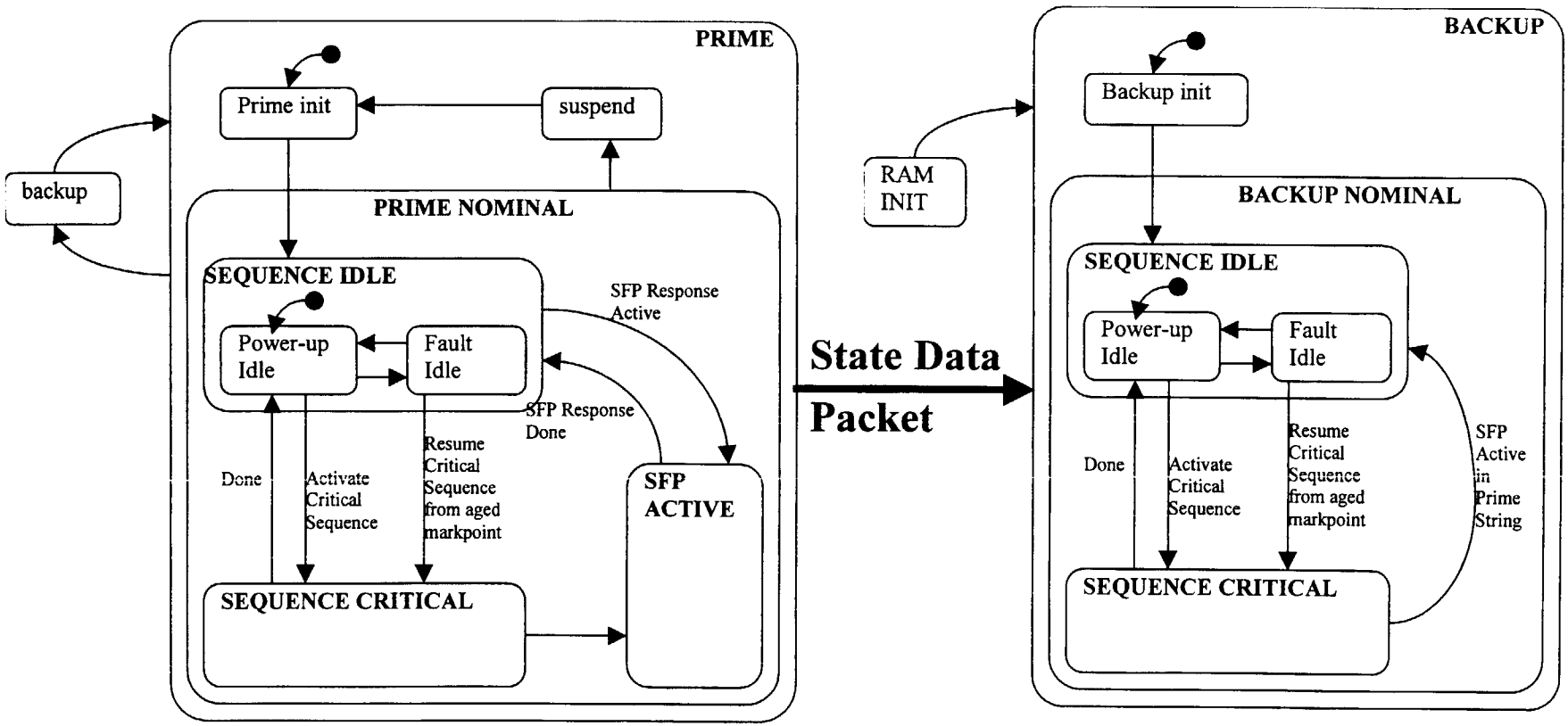
1. **Three seconds of aging for completed subsequences**

Overall redundancy allows entire controlling subsystem to fail:

1. **Failure is detected via communication mechanism**
2. **Backup system becomes Control & repairs fault**
3. **New Control system rolls back and resumes execution**

Example: Sequence execution segment:





Design Validation:

- **Original state space without design abstraction contained about 2^87 states.**
- **Partitioned requirements into 5 equivalence classes**
- **Validated 6 Checkpoint requirements**
- **Reduced state space to about about 130, 000 states**
- **Found 3 Anomalies using SPIN Model Checker [Gerard Holzman, Bell Labs, N.J.]**
- **Run Time was about 30 seconds for each anomaly:**

1. Repeated prime failure caused loss of synch with backup
2. Prime failure immediately at end of sequence resulted in no rollback
3. Prime failure at second 2 after check point gave invalid rollback point in backup system

Implementation Validation:

1. **Used test harness for validation vehicle**
2. **Language is Tool command Language (Tcl)**
3. **Specify sequence to execute**
4. **Inject design faults**
5. **Select data structures to output for state vector sequence to examine**
6. **Look for design faults in output software implementation state vector sequence**

Implementation Validation of Design Anomalies

1. We did not see the first anomaly in the system - system engineers noted that all faults take at least several minutes to repair. Therefore , repair time was extended so that anomaly one would not be seen

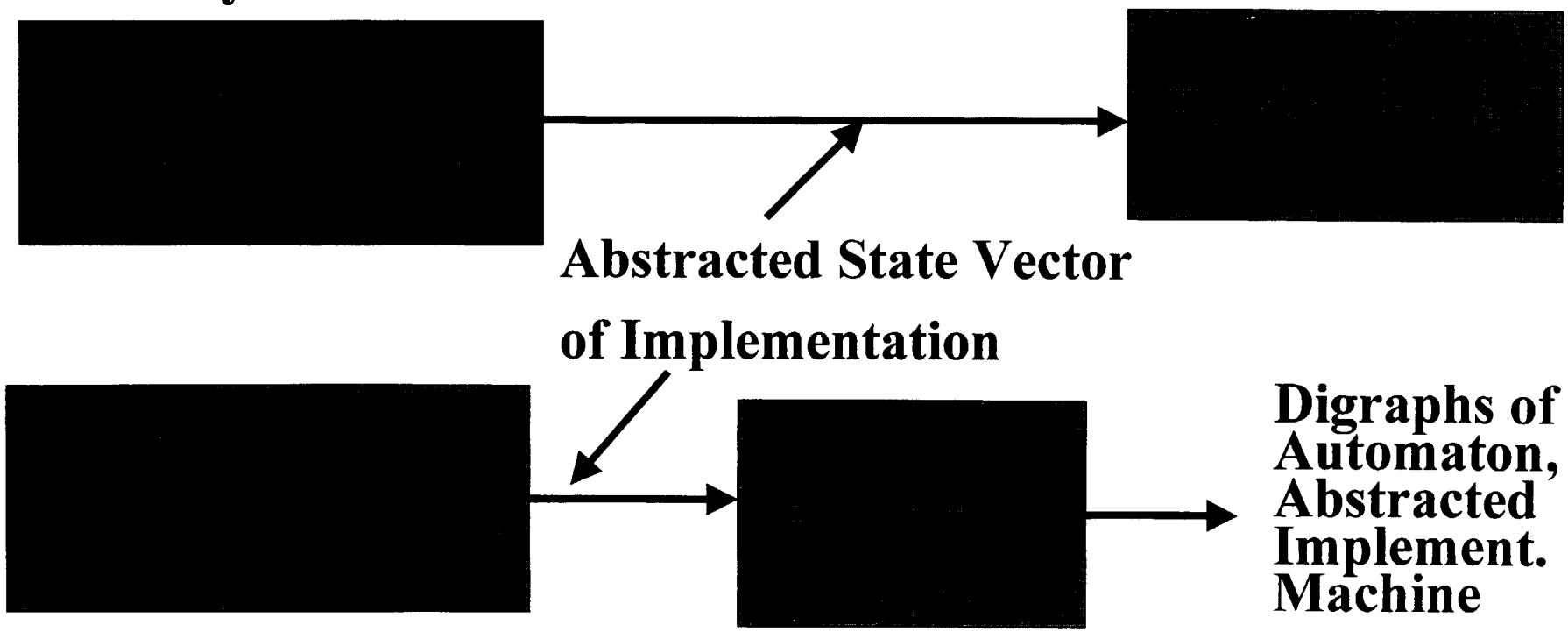
2. Our technique demonstrated that the second anomaly was in the implementation.

3. Our technique demonstrated that the third anomaly was in the implementation.

4. Project is addressing the issues of the three anomalies

Automation of Implementation:

1. Exactly as above but with an automaton replacing human scan of output state vector sequence
2. Automaton accepting state corresponds to counter example: If automaton is driven into accepting state requirement is violated.
3. Anomaly three was validated with automaton



Future work:

1. Replace Test harness output by appropriately abstracted state vector from executing spacecraft application
2. Develop counterexample requirements automaton for currently executing application
3. Drive automaton from abstracted state vector in 1.
4. Use accepting state condition of automaton as fault detection criterion to start fault recovery scheme.

Benefits:

1. Puts current fault detection and recovery schemes into analytic framework
2. Counterexample automaton could be derived on-the-fly for autonomous spacecraft applications

Tying it all Together

Having shown that we could

- **Abstract a high level design from a specification for a spacecraft**
- **Validate the design using model checking**
- **Validate the implementation**

We decided to incorporate the model checking process into our Quality Assurance processes:

- **Currently there are about 21 processes.**
- **Process chart is flow chart that specifies logistics to follow in carrying out a quality assurance process for a quality process such a formal methods theorem proving, model checking, software inspection, and the like**
- **Process Chart is accompanied by written description**

Using Model Checking to Validate AI Planner Domain Models

John Penix, Charles Pecheur and Klaus Havelund
Automated Software Engineering Group
NASA Ames Research Center
M/S 269-3
Moffett Field, CA 94035
jpenix,pecheur,havelund@ptolemy.arc.nasa.gov

Abstract

This report describes an investigation into using model checking to assist validation of domain models for the HSTS planner. The planner models are specified using a qualitative temporal interval logic with quantitative duration constraints. We conducted several experiments to translate the domain modeling language into the SMV, Spin and Murphi model checkers. This allowed a direct comparison of how the different systems would support specific types of validation tasks. The preliminary results indicate that model checking is useful for finding faults in models that may not be easily identified by generating test plans.

1 Introduction

In the classical approach to analyzing the correctness of a piece of software is broken into two tasks: verification and validation. Verification is the task of making sure that the software implementation complies with a stated set of requirements. Validation is making sure that the stated requirements correctly reflect the needs of the end user.

In the realm of artificial intelligence and knowledge-based systems, the role of validation shifts slightly. These systems are generally divided into two parts: a knowledge base that is used as a model of the environment with which the program interacts, and a reasoning algorithm that manipulates the knowledge base during program execution. The accuracy of the knowledge base with respect to the real environment has direct implications on the performance of the AI system. A system with a totally correct reasoning algorithm will be ineffective if its model of the world is flawed. Therefore, validation becomes a critical task of evaluating a part of the system to be deployed.

We are currently investigating the use of model checking [2, 4, 5] to assist in the validation of models used in the HSTS Planner [6], a model-based planning system that is being used in the Remote Agent autonomous control system architecture [7]. The planner takes an initial state and a

goal as inputs and produces a plan for achieving that goal. The planning algorithm guarantees that the generated plan is consistent with a set of temporal constraints that model physical limitations of the controlled system and the environment.

Because domain models are often composed of a large number of tightly coupled constraints, the interactions among constraints can become conceptually unmanageable very quickly. Therefore, it is desirable to have methods to validate the model by finding inconsistencies in the model and determining whether implicit properties of the model can be derived from the set of explicit constraints.

2 Constraint Model Specifications

The HSTS Planner domain models are described using the HSTS Domain Description Language (DDL), an object-oriented constraint specification language based on Allen's temporal interval logic [1].

DDL models define object classes and instances. Each object encapsulates a number of state variables. A state variable is declared to be *controllable* if it can be modified during scheduling and *uncontrollable* otherwise. For example, a robot object class could be defined and instantiated as follows:

```
(Define_Object_Class Robot
  :state_variables
  ((Controllable Task)
   (Controllable Location)))

(Define_Object Robot Robbie)
```

The language provides special support for the definition of objects without any properties as sets of labels:

```
(Define_Label_Set Room (Kitchen Hallway LivingRoom))
```

A set of *member values* is defined for each state variable. The member values are called *predicates* and may have parameters. The name of a predicate and the value of its parameters define the value of a state.

Predicate parameters can be objects, labels or built in types. For example, possible values for the Task and Location state variables in the Robot class are defined in Figure 1. In this example, the Task state variable for the Robot class can either has the value `Moving(`x`,`y`)`, where x and y are locations, or it has the value `Idle`. The Location state in the Robot class can have the value `In_Room(`x`)`, where x is a location.

```
(Define_Predicate Moving
  ((Room From)
   (Room To)))

(Define_Predicate Idle)
(Define_Member_Values ((Robot Task))
  (Moving Idle)

(Define_Predicate In_Room
  ((Room R)))
(Define_Member_Values ((Robot Location))
  (In_Room))
```

Figure 1: Example Predicate and Member Value Definitions

Temporal Relation	**Inverse Relation**	**Endpoint Relation**
T1 **before** (d D) T2	T2 **after** (d D) T1	$d \leq$ T2.start - T1.end $\leq D$
T1 **starts_before** (d D) T2	T2 **starts_after** (d D) T1	$d \leq$ T2.start - T1.start $\leq D$
T1 **ends_before** (d D) T2	T2 **ends_after** (d D) T1	$d \leq$ T2.end - T1.end $\leq D$
T1 **starts_before_end** (d D) T2	T2 **ends_after_start** (d D) T1	$d \leq$ T2.end - T1.start $\leq D$
T1 **contains** ((a A) (b B)) T2	T2 **contained_by** ((a A) (b B)) T1	$a \leq$ T2.start - T1.start $\leq A$ $b \leq$ T1.end - T2.end $\leq B$
T1 **parallels** ((a A) (b B)) T2	T2 **paralleled_by** ((a A) (b B)) T1	$a \leq$ T2.start - T1.start $\leq A$ $b <$ T2.end - T1.end $< B$

Table 1: HSTS DLL Temporal Relations

A DDL model is constrained by defining compatibility conditions that must hold between different values of state variables. The constraints are specified in terms of temporal intervals (called *tokens*) during which a sate variable holds a specific value or set of values. The compatible relationships between tokens are specified using a set of predefined temporal operators. The operators permit the expression of all possible temporal relationships between the start and end points of two tokens. The DDL temporal relationships are shown in Table 1. DDL also supports abbreviations for common temporal relations as shown in Table 2.

DDL compatibility specifications are defined using *token descriptors*. A token descriptor is a pattern that describes attributes of a set of tokens. These patterns can refer to either one (SINGLE) or a sequence of tokens (MULTIPLE). Compatibility specifications have the form:

(*master-token-descriptor compatibility-tree*)

Abbreviation	Temporal Relation
before	**before** (0 $+\infty$)
after	**after** (0 $+\infty$)
meets	**before** (0 0)
met_by	**after** (0 0)
starts	**starts_before** (0 0) = starts_after (0 0)
ends	**ends_before** (0 0) = ends_after (0 0)
contains	**contains** ((0 $+\infty$) (0 $+\infty$))
contained_by	**contained_by** ((0 $+\infty$) (0 $+\infty$))
equal	**contains** ((0 0) (0 0))

Table 2: Abbreviations for Common Temporal Relations

where *compatibility-tree* is an and/or tree of pairs of temporal relations and token descriptors. For example, we can specify that the robot can only move from the Kitchen to the LivingRoom by going through the Hallway as follows:

```
(Define_Compatibility
 (SINGLE ((Robot Task)) ((Moving (Kitchen LivingRoom))))
 :compatibility_spec
 (AND (met_by (SINGLE ((Robot Location)) ((In_Room(Kitchen)))))
      (equals (SINGLE ((Robot Location)) ((In_Room(Hallway)))))
      (meets (SINGLE ((Robot Location)) ((In_Room(LivingRoom)))))))
```

In addition, variables can be used to constrain parameter values between the master token descriptor and the token descriptors in the compatibility tree.

3 Model Checking for DDL

For the purpose of translating HSTS models into suitable inputs for model checkers, we need to express HSTS models in terms of states and transitions between states. In HSTS, the state of the system is defined by the values of all state variables of all objects. A transition occurs when the value of at least one state variable changes.

For describing what happens upon a transition, we take the convention that V (resp. V') denotes the value of variable V before (resp. after) the transition. We then define the following abbreviations for convenience:

$$\texttt{(SV to P)} \equiv \texttt{(SV != P) AND (SV' == P)}$$

$$\texttt{(SV from P)} \equiv \texttt{(SV == P) AND (SV' != P)}$$

DDL Constraint	State Translation Relation
((SINGLE (SV1 P1)) meets (SINGLE (SV2 P2)))	(SV1 from P1) ⇒ (SV2 to P2)
((SINGLE (SV1 P1)) met_by (SINGLE (SV2 P2)))	(SV1 to P1) ⇒ (SV2 from P2)
((SINGLE (SV1 P1)) starts (SINGLE (SV2 P2)))	(SV1 to P1) ⇒ (SV2 to P2)
((SINGLE (SV1 P1)) ends (SINGLE (SV2 P2)))	(SV1 from P1) ⇒ (SV2 from P2)
((SINGLE (SV1 P1)) contained_by (SINGLE (SV2 P2)))	((SV1 == P1) ⇒ (SV2 == P2)) AND ((SV1' == P1) ⇒ (SV2' == P2))
((SINGLE (SV1 P1)) equals (SINGLE (SV2 P2)))	(SV1 to P1) ⇒ (SV2 to P2) AND (SV1 from P1) ⇒ (SV2 from P2) AND ((SV1 == P1) ⇒ (SV2 == P2)) AND ((SV1' == P1) ⇒ (SV2' == P2))

Table 3: Translation from DDL Constraints to State Transition Computation Model

Using these, some of the abbreviated temporal relations from Table 2 can be expressed as a logical relation that describes legal transitions between states. Table 3 shows the translation for six of the basic DDL qualitative relations.

Some of the translations can be further simplified. For example, the contained_by translation can be simplified to

$$((SV1' == P1) \Rightarrow (SV2' == P2))$$

plus checking that in the initial state

$$((SV1 == P1) \Rightarrow (SV2 == P2))$$

This simplification was used in the SMV model.

This generic translation was used as the basis for translation into the input languages for 3 different model checkers: Spin [5], SMV [2] and Murphi [4]. The translation had to be done slightly differently for each due to differences in their input languages. For example, the translation into both Spin and Murphi includes a small program that selects the next state of the model based on the state transition relation. In SMV, the input languages allows the specification of a state transition relation as a propositional formula, simplifying the translation.

4 Results

We tested our translation on a small model of an autonomous robot. The model consisted of a battery powered robot that could move between three locations: A, B, and In_Between. The goal of the robot was to fix a hole at location A. The model had 65 temporal constraints which allowed 16320 reachable states out of a potential 559872 states. The constrains in the model dictated such things as:

```
((Robot.Task = Moving) contained_by (Hole.Charge = Charge_Full))
  ((Robot.Task = Fixing_Hole) meets (Hole.Status = Hole_Fixed))
```

The analysis identified several potential flaws in the model that were not identified during testing.

4.1 Expressibility

The subset of the language that is covered by the initial translation is expressive enough to cover the majority of the modeling done for the robot. We were not able to capture a couple of quantitative durations that were used in the model. The inability to translate nonlocal constraints was not a limitation in this example.

4.2 Analysis Capabilities

We analyzed the model for both safety and liveness properties. Safety properties state that nothing wrong ever happens, e.g. that no deadlock occurs. They are simple to verify: the model checker checks if any reachable state violates the property. Liveness properties state that something good must eventually happens, e.g. that a query is always answered. They are more complex to handle because they apply to runs rather than states. Some model checkers, like Murphi, do not support liveness properties; others, such as SPIN and SMV, use a more elaborate state space exploration algorithm to verify them.

4.2.1 Safety Properties

Safety properties can be used to check whether plans exist within a domain model. This is done by checking whether there is a path that leads to a goal state, or, cast as a safety property, checking if it is never the case that the system reaches a goal state. If there is such a case, the model checker will return a series of state transitions that will lead to the goal state, which is in essence a plan.

For example, in SMV we could ask if there was a plan where the robot fixes the hole using the query

```
!EF (Hole.Status = Hole_Fixed)
```

which reads, "for all initial states, there does not **E**xist a **F**uture state where the hole is fixed." In an early version of the model, this analysis yields "true", meaning that the hole could never be fixed. Further queries revealed the fact that the robot could not move. This was because of an overly general constraint that was intended to constrain movement to and from locations A and B, but also ended up constraining location In_Between. The fix was to replace the general constraint with individual constraints for A and B. After the fix, analysis of the above specification gave a sequence of states that lead to the hole being fixed.

There were several other queries that were useful for inspecting the model. First, it was interesting to see if a plan existed from *any* initial state in the model, not just *some* initial state as is

the case above. This can be done with the query

```
EF (Hole.Status = Hole_Fixed)
```

which reads, "for all initial states, there **E**xists a **F**uture state where the hole is fixed." Note that this specification is not the logical opposite of the previous query, because SMV includes an implicit quantification over all legal initial states, requiring "for all initial states" to be read before any query. Analysis of the second specification gives an example of a initial state from which the hole cannot be fixed, i.e. one where the robot starts away from a recharge station, with no charge. This analysis therefore useful for identifying initial states that may have been unintended.

4.2.2 Liveness Properties

In the robot model, a desirable liveness property is that there *always* exists a sequence of states that leads to the hole being fixed. This can be tested using the query

```
AG EF (Hole.Status = Hole_Fixed)
```

which asks, "for all initial states, and for all states (**G**lobally) on **A**ll paths, there **E**xists a path with a **F**uture state where the hole is fixed." Analysis of this specification revealed that it was not true. An error trace was reported in which the robot fixes the hole and then the hole reappears. This leaves us in the situation where the robot is uncharged away from a recharge station and cannot fix the new hole. If the model is constrained such that a fixed hole stays fixed, then this specification becomes true.

Another liveness property is that the hole eventually gets fixed. That is, "**A**ll paths lead to a **F**uture state where the hole is fixed. In SMV, we can write this as:

```
AF (Hole.Status = Hole_Fixed)
```

Without further hypotheses, it turns out that this formula does not hold. Indeed, SMV reports a diagnostic trace where the robot remains idle forever, which is a valid behavior according to the model but unlikely to be chosen by the more proactive HSTS planner.

To obtain a more meaningful result, we consider only "fair" runs, such that no enabled move is ignored indefinitely. The property that all such fair runs eventually reach a property p is captured by the SMV formula `!E[!p U (AG !p)]` [8], so the formula above becomes:

```
!E[!(Hole.Status = Hole_Fixed) U AG !(Hole.Status = Hole_Fixed)]
```

which means that, "from all initial states, there does not **E**xist a path where the hole remains not fixed **U**ntil a point where the hole is never (**AG!**) fixed." SMV indeed reports this to be a valid property of our model.[1]

[1] SMV also has a FAIRNESS declaration for declaring application-specific fairness constraints explicitly, which has not been used here.

4.3 Performance

For the initial subset of the language addressed, the SMV model checker is much faster than its competitors (0.05 seconds vs. 30 seconds). There were two reasons for this result. The main reason is that the reachable traces through the system were shallow relative to the size of the state space. This type of model is more amiable to the symbolic state space representation and searching technique employed by SMV. This method represents sets of states as logical predicates avoiding the need to represent each state explicitly. The second reason is that SMV allows explicit specification of a state transition relation as opposed to the standard guarded command language or programming language format for model checkers. This allowed the translated DDL constraints to be mapped directly into the model checking language without the need for additional mechanisms to select the next state based on the constraints. Further studies will investigate whether SMV can maintain this advantage when the translation is expanded to handle non-local constraints and quantitative intervals.

5 Related Work

The link between planning and model checking is not new; Cimatti et al [3] use the SMV model checker as a reasoning engine to do planning. There work has the same spirit as ours because it also requires translation from a planning language to a model checking language. However, the differences between the two planning languages are significant, making the technical details dissimilar.

6 Conclusion

Our initial investigations show that model checking can be effective at finding flaws in HSTS domain models. The exhaustive search capabilities of model checking will discover modeling errors that may be overlooked by the heuristic search done during planning. Our future work will involve expanding the expressiveness of the language covered by the translation and, after determining the best target, automating the translation so that model checking can be used directly by domain experts.

Acknowledgements

We would like to thank Nicola Muscettola and Kanna Rajan for their help in understanding DDL and the HSTS planner, Lina Khatib for allowing us to use her robot model, and Nikolaj Bjorner for his enlightening discussions about the performance of various model checkers on the robot example.

References

[1] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10e20 states and beyond. In *Proceedings of The International Conference on Logic in Computer Science*, 1990.

[3] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$. In S. Steel and R. Alami, editors, *Proceeding of the Fourth European Conference on Planning*, number 1348 in Lecture Notes in Artificial Intelligence, pages 130–142, Toulouse, France, September 1997. Springer-Verlag. Also ITC-IRST Technical Report 9705-02, ITC-IRST Trento, Italy.

[4] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[5] Gerard J. Holzmann. *Design and Verification of Protocols.* Prentice Hall, 1990.

[6] Nicola Muscettola. Hsts: Integrating planning and scheduling. In M. Zweben and M. S. Fox, editors, *Intelligent Scheduling.* Morgan Kaufmann, 1994.

[7] Barney Pell, Erann Gat, Ron Keesing, Nicola Muscettola, and Ben Smith. Plan execution for autonomous spacecraft. In *Proceedings of the 1997 International Joint Conference on Artificial Intelligence*, 1997.

[8] J. P. Queille and Joseph Sifakis. Fairness and related properties in transition systems - a temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983.

Using Model Checking to Validate AI Planner Domain Models

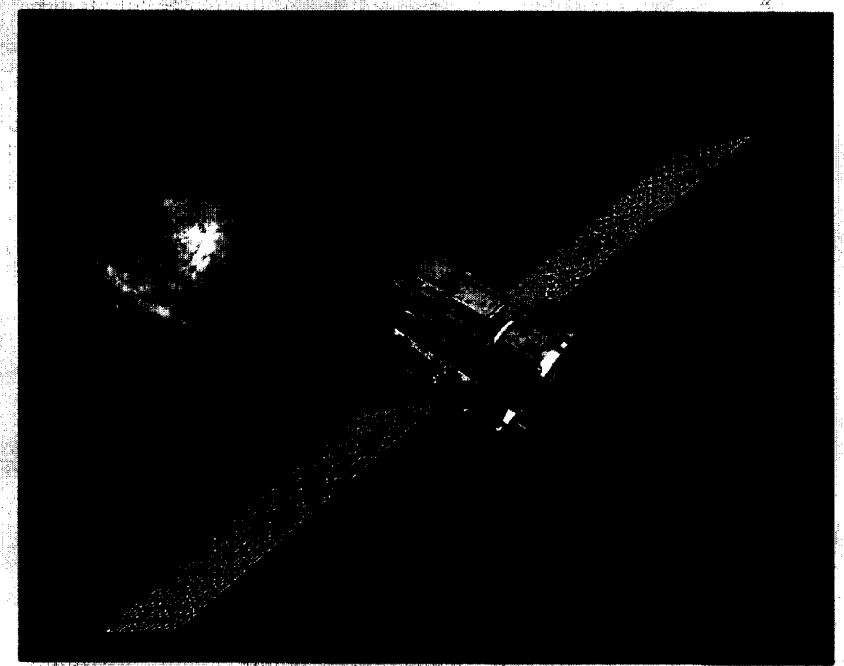
John Penix, Charles Pecheur and Klaus Havelund

Automated Software Engineering

NASA Ames Research Center

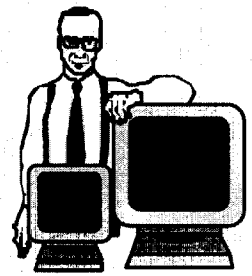
The Problem: High-Assurance Autonomous Systems

- **Long lifetimes**
- **Limited Access**
  - **Reactive/Adaptive**
  - **Agent-Based Architectures**
  - **Advanced Algorithms**
  - **Knowledge-Based**



Verification and Validation of Autonomous Planning Systems

Our approach is to separate algorithm verification from domain model validation



Computer Scientists

Domain Engineers

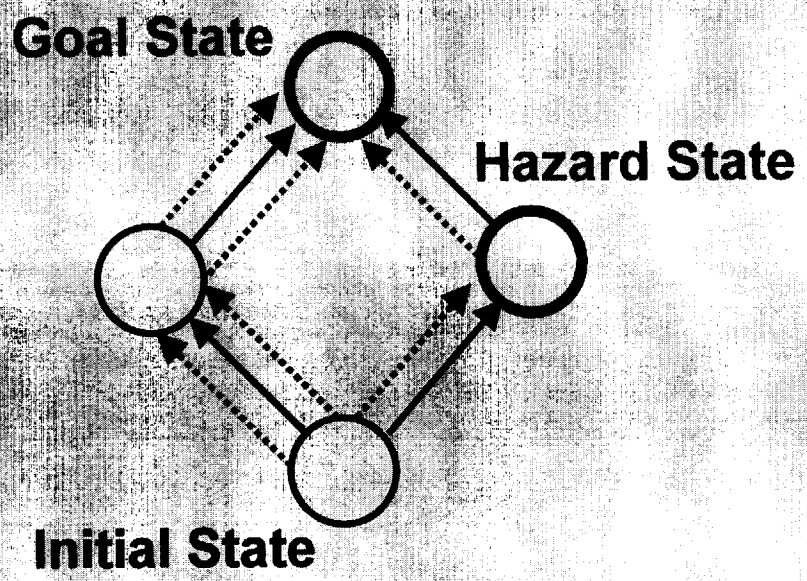
Autonomous Planner

Model Checking

- *Technique for verification of finite-state systems - traditionally used for hardware*
- *Determines whether a FSM is a "model" of a temporal logic formula*
  - *Exhaustive - evaluates **all** possible executions of events in the system*
  - *Allows non-deterministic modeling/abstraction of the "environment"*
  - *Limited by "state space explosion"*

Model Checking vs. Planning

***Exhaustive** model checking algorithms can find flaws in domain models that may not be discovered by testing with a **heuristic** planner*



The planner may find a path to the goal without discovering the hazard state. Model checking tries all paths and finds the hazard state.

The HSTS Planner's Domain Description Language

Object-oriented data structures with qualitative and quantitative constraints on variable values

Robot
:state variables
At: {A,B,C}
Task: {Move,Fix,Rest}
Charge: {Empty,Full}

Hole
:state variables
At: {A,B,C}
Status: {Exists,Fixed}

((Robot.Task=Fix) starts_before (10 20) (Hole.Status = Fixed))

Model Checking for DDL

- We developed a translation from a DDL model to a finite state transition model
- Tested the translation on 3 model checkers:
  - Spin (from Bell Labs)
  - Murphi (from Stanford University)
  - SMV (from Carnegie Mellon University)

Results: Expressibility

General translation to finite state transition model able to express qualitative constraints with temporal locality

- *No support for quantitative constraints*
- *Temporally distant relationships require special consideration:*
  - *history variables in Spin & Murphi*
  - *fairness constraints in SMV*

Results: Performance

- Experimented on "small" model of autonomous robot with 65 temporal constraints
- The model had 16320 reachable states out of 559872 potential states (highly constrained)
- Exhaustive verification with Spin & Murphi in under 30 seconds
- Exhaustive verification with SMV in 0.05 seconds (due to better target language)

Results: Capabilities

Model checking "error traces" are plans

Able to perform analysis that are not directly supported by testing with the planner:

- *Is there a plan (a path from an initial state to a goal state) for **any** legal initial state?*
- *Is there a plan for **every** legal initial state?*
- *Is there a plan from every reachable state?*
- *Can I reach a state where X is true?*

Conclusions

- Model checking has the potential to overcome limits to model validation using a planner
- Model checking can be used to effectively validate "simple" domain models
- Further experiments are necessary to see if temporally distant relations and quantitative constraints can be effectively supported

V&V of a Spacecraft's Autonomous Planner through Extended Automation

Martin S. Feather
Jet Propulsion Laboratory,
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109, USA
+1 818 354 1194
Martin.S.Feather@Jpl.Nasa.Gov

Ben Smith
Jet Propulsion Laboratory,
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109, USA
+1 818 353 5371
Ben.D.Smith@Jpl.Nasa.Gov

ABSTRACT

We have introduced and used significant automation during the verification and validation (V&V) of a spacecraft's autonomous planner. This paper describes the problem we faced, the solution we employed, and the applicability of our approach in a general V&V setting.

PROBLEM

Cost, performance and functionality concerns are driving a trend towards use of self-sufficient autonomous systems in place of human-controlled mechanisms. Our focus has been the verification and validation (V&V) of a spacecraft's autonomous planner. This planner generates the sequences of high-level commands that control the spacecraft. The planner is part of a self-sufficient autonomous system that will operate a spacecraft over an extended period, without human intervention or oversight. Hence, V&V of the planner is crucial.

The planner can exhibit a much wider range of behaviors that the command sequence mechanisms of more traditional spacecraft designs. Furthermore, it must respond correctly to a wide range of circumstances. Together, these raise some new challenges for V&V.

As for any complex piece of software, a major focus of V&V revolves around thorough testing. The new V&V challenges manifest themselves during testing as the following combination of characteristics:

- The planner's output (plans) are detailed and voluminous, ranging from 1,000 to 5,000 lines long. Plans are intended to be read by software, and are not designed for easy perusal by humans. To illustrate this, a small fragment of a plan is shown in Figure 1.
- Each plan must satisfy all of the flight rules that characterize correct operation of the spacecraft. Flight rules may refer to the state of the spacecraft and the activities it performs, and describe temporal conditions required among those states and activities. Flight rules are expressed in a special-purpose language; an example is shown in Figure 2. There are over 200 such flight rules of relevance to the planner.
- The information pertinent to deciding whether or not a plan passes a flight rule is dispersed throughout the plan.
- The thorough testing of the planner yields thousands of such plans, spanning the wide range of circumstances in which the planner is expected to operate.

As a consequence, manual inspection of more than a small fragment of plans generated in the course of testing is impractical.

SOLUTION

Our approach has been to automate the checking of plans. The automated system checks each plan for adherence to all of the flight rules input to the planner. This verifies that the planner is not

```
(#S(C-TOKEN
   :CARDINALITY :SINGLE   :NAME VAL-920
   :SV-SPEC (SPACECRAFT_ATTITUDE  SPACECRAFT_ATTITUDE_SV)
   :TYPE-SPEC ((CONSTANT_POINTING_ON_SUN
                  (HGA_AT_EARTH  BBC_DEADBAND_CRUISE)))
   :START-B-TOKEN VAL-920
   :END-B-TOKEN VAL-920
   :STATE-VARIABLE (SPACECRAFT_ATTITUDE SPACECRAFT_ATTITUDE_SV)
   :TOKEN-TYPE ((CONSTANT_POINTING_ON_SUN
                    (HGA_AT_EARTH   BBC_DEADBAND_CRUISE)))
   :DURATION (37801 500000000)
   :START-TIME-POINT TP-1279
   :END-TIME-POINT TP-1116
```

Figure 1 – Small fragment of a plan

Every interval of SEP_Thrusting *whose* 4^{th} *parameter* = FIRST *is* "contained_by" *an interval of* Sun_Pointing *with the same* 1^{st} *parameter as the* 1^{st} *parameter of the thrusting interval, and with* 2^{nd} *parameter* = BBC_DEADBAND_IPS_TVC

```
(Define_Compatibility
   (SINGLE ((SEP SEP_SV))
            ((SEP_Thrusting ( ?heading ?level ?duration FIRST))))
  :compatibility_spec
  (contained_by
   (SINGLE ((Spacecraft_Attitude Spacecraft_Attitude_SV))
            ((Sun_Pointing ( ?heading BBC_DEADBAND_IPS_TVC))))
```

Figure 2 – Example flight rule

generating hazardous command sequences. The automated system also performs some validation checks. These arise from a gap between the "natural" form of a flight rule, and the way in which it must be re-encoded so as to be expressed to the planner. The automated system checks a direct encoding of the "natural" statement of the flight rule, thus helping validate that the planner and its inputs are accomplishing the desired behavior.

We use a database as the underlying reasoning engine of our system to automatically check plans. To perform a series of checks of a plan, we automatically load the plan as data into the database, having previously created a database schema for the kinds of information held in plans. We express the flight rules as database queries. The database query evaluator is used to automatically evaluate those queries against the data. Query results are organized into those that correspond to passing a test, which we report as confirmations, and those that correspond to failing a test, which we report as anomalies.

The net result is that we can quickly and thoroughly check each plan. The automated checking code takes less than five minutes (on a Sun ULTRA Sparc) to perform each of several hundred checks of a large (5,000 line) plan and generate a report of the results. Plan generation is a search-intensive activity, and a planner is a

complex piece of software precisely because of the need to perform this search in an effective and efficient manner. Conversely, once a plan has been generated, checking properties of that plan is relatively straightforward.

Because the flight rules themselves are numerous and detailed, and evolve over the course of software development, we have taken the automation one step further. We generate the verification part of the plan-checking code from the flight rules themselves, in the same form in which they are input to the planner. Using this capability, we are able to *automatically* regenerate the flight-rule checking code, whenever the set of flight rules input to the planner evolves. The architecture of this system is shown in Figure 3.

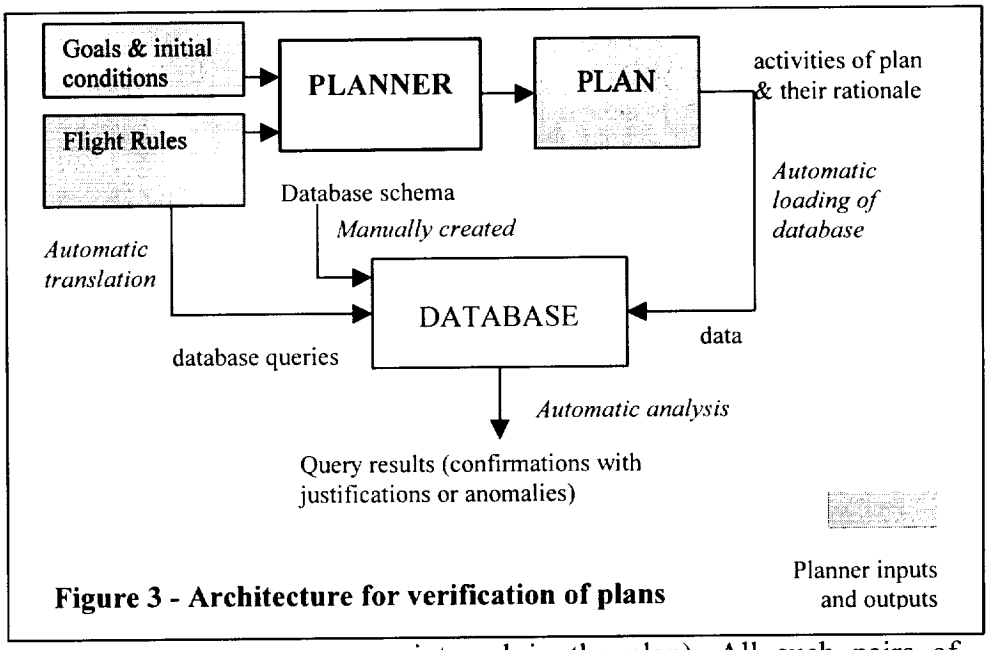


Figure 3 - Architecture for verification of plans

The pieces we had to build were:

- The database schema to hold plan information.
- Code to automatically load a plan (in the form output by the planner) into the database.
- Code to automatically generate a report from running the database queries. A report contains more than simply a pass/fail result for the plan as a whole. For example:
  - Flight rules that are satisfied trivially are reported as such (e.g., the flight rule shown in Figure 2 would be trivially satisfied if the plan contained no intervals of SEP_Thrusting).
  - Flight rules that are satisfied by finding corresponding activities in the plan are reported as such (e.g., the flight rule shown in Figure 2 would be satisfied by finding a Sun_Pointing interval in the plan corresponding to an SEP_Thrusting interval in the plan). All such pairs of corresponding intervals are reported.

  This kind of information is useful to the planning team in assessing test coverage.
- Code to automatically translate flight rules (in the form input to the planner) into database queries.

METRICS

The checker tool has been used during of the testing of the spacecraft's autonomous planner.

- It is applied to check every flight rule input to the planner. There over 200 such rules.
- It is applied to the plans generated during testing. To date, there have been thousands of such plans.
- The checker runs somewhat faster than the planner; the time to check a plan typically ranges from 30 seconds to 4 minutes, while the time to generate a plan typically ranges from 3 minutes to 10 minutes.
- When there is a change to the flight rules, we automatically regenerate the checker's

database queries. This takes less than 10 minutes for the entire set of flight rules. Complete regeneration, in response to flight rule changes, has been performed 3 times.

- The development of the checker was a significantly lesser effort than the development of the planner. The former took several months, the latter several years.
- The checker was modified to accommodate a modest change to the plan syntax. This took less than 3 days to accomplish. A small change to the syntax of the flight rules was accommodated in less than one hour.

APPLICABILITY

Our approach has been developed for, and applied to, V&V of a spacecraft's autonomous planner. However, we believe the approach has much wider applicability than this one project. The characteristics that identify when this approach is worthwhile and viable are as follows:

Worthwhile: The development of automated test checking code, rather than relying upon manually conducted checks, is warranted when:

- There are voluminous amounts of data to check, either because each test run yields lots of data, or there are numerous test runs, or both.
- The checking of a test run is complex, either because there are many checks to perform, or the checks themselves are hard to perform, or both.

These conditions render manual checking unsatisfactory.

A further applicability condition is that it is infeasible to analyze the code itself in place of testing the code. For our task, the planner was a complex piece of software, and seemed beyond the capabilities of present-day analysis techniques (such as model checking or theorem proving). This rendered thorough testing, and therefore thorough checking of the test results, inevitable.

Viable: The style of automated checking we developed requires the following conditions to hold:

- The data to check is self-contained. That is, there is no need for human interaction to determine whether or not a check has been met. (In our planner task, each plan is a self-contained object from which it can be determined whether or not each flight rule holds.)
- The data to check is in a machine-manipulable form. That is, it is feasible to develop automated checking that will work directly off the form of data available, without human intervention. (In our planner task, plans have exactly this characteristic, since they are intended for consumption by the spacecraft's automatic executive.)
- Checking is easier than generation. That is, the code to check that a test run satisfies the desired conditions is simpler than the code that generates that test data.

  This has two positive consequences:

  1. The development of the automated test checking code will be a much lesser effort than the development of the system under test.
  2. The test checking code will run faster than the system under test (meaning it can easily keep up with the test data generated, and provide quick feedback to the test personnel).

Our automatic generation of flight-rule checking code reflects the same characteristics of an activity that is worthwhile and viable to automate:

- we have hundreds of flight rules to check
- individual rules can be quite complex
- the set of rules evolves over time

- flight rules are expressed in a machine-manipulable format (constraints input to the planner)
- the language of those rules (planner constraint language) is carefully proscribed so as to render plan generation feasible; the expression of those rules as checks can employ an extensible, general purpose language.

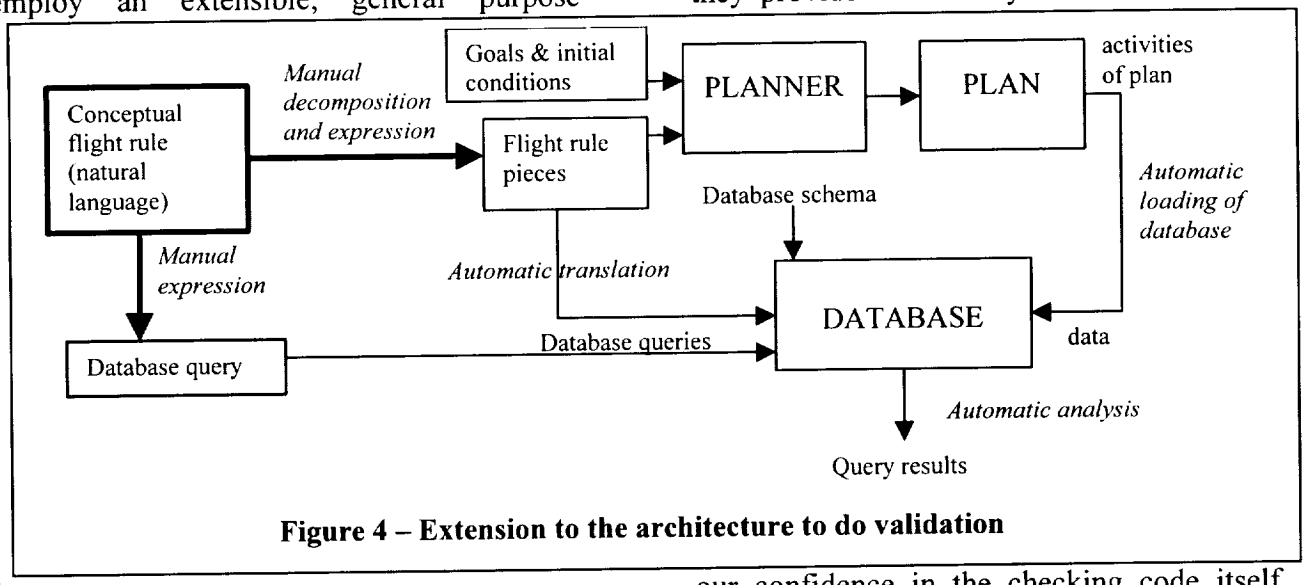


Figure 4 – Extension to the architecture to do validation

In our system, generation of the flight-rule checking code takes under 10 minutes and is completely automatic.

FURTHER OBSERVATIONS

Our problem and solution exhibit two further characteristics of general importance.

The value of redundancy and rationale: Each plan generated by the spacecraft's planner contains both a sequence of activities, and justifications for those activities. These justifications relate each activity to the flight rules that were taken into account in planning that activity. Viewed solely as a command sequence, the presence of these justifications in the plan is redundant. However, these justifications serve two very useful roles for V&V purposes:

- they provide rationale for why the planner arrived at a plan. This rationale can be checked to ensure that the planner is not only arriving at the "right" solution (namely, a plan that adheres to all the flight rules), but is doing so for the "right" reasons. This gives the test team confidence to extrapolate the correct operation of the planner to a wide range of circumstances.
- they provide redundancy that contributes to our confidence in the checking code itself. Our test checking code independently performs the following three kinds of checks:

1. that the activities of the plan adhere to all the flight rules,
2. that there is a justification recorded with each activity in the plan for every flight rule that the checker finds is applicable to that activity, and
3. that every justification recorded in the plan can be traced back to a flight rule.

This makes it unlikely that the checking code has a "blind spot" that happens to overlook a fault in a plan.

The automated test checking code we automatically generate from planner flight rules checks this rationale.

Opportunities for validation: Verification was the original focus of our plan checker generation

effort. By thorough checking of the planner's outputs (plans) against the flight rules given as input to the planner, we gained confidence that the internal operation of planner was correct. However, the effort also yielded significant opportunities for validation.

Validation opportunities arose from a gap between the most "natural" statement of a flight rule, and the form in which it must be re-encoded so as to be expressed to the planner. The planner constraint language is carefully proscribed so as to render plan generation feasible. On occasion, a flight rule cannot be expressed directly in this limited language. Instead, it must be (manually) subdivided into several separate rules that in conjunction will achieve the requisite condition, and that individually can be expressed in the constraint language. Our language for expressing checks is more general purpose than the planner constraint language. This means that it is possible to (manually) encode an automatic check corresponding directly to the original flight rule. By following this process, we are able to validate that the planner, and the encodings of flight rules given to it, do in fact achieve the original intent.

Note that there is a manual step to this validation - we must manually encode the original flight rules (expressed in natural language) as checking code. The checking code then runs automatically. However this manual step can take advantage of the framework established by the verification architecture and code.

In more general terms, we see that verification can be extended into the realm of validation when the verification language is more general than the language of the system being verified.

CONCLUSIONS

Testing activities are an area ripe for insertion of automation. Our work automates the determination of whether a test run has met its requirements. Furthermore, we automate the generation of the code performing these determinations. We were motivated in part by early work in this direction, reported in [1].

We employ a database at the heart of our checking tool. Our earlier pilot studies had shown a database could be used to provide rapid and flexible analysis [2].

Checking test runs is only a part of testing. For example, selecting *which* tests to run is an important decision. Other than providing some feedback on which requirements a test run has exercised, the work reported here does not address test selection. For a broader perspective on the testing of autonomous spacecraft software, see [3].

ACKNOWLEDGMENTS

The research and development described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space administration. Funding was provided under NASA's Code Q Software Program Center Initiative UPN #323-098-5b, and by the Autonomy Technology Program.

REFERENCES

[1] D.J. Richardson, S.L. Aha & T. O'Malley, "Specification-based Test Oracles for Reactive Systems," Proceedings of the 14th International Conference on Software Engineering, pp. 105-118, Melbourne, Australia, 1992.

[2] M.S. Feather, "Rapid Application of Lightweight Formal Methods for Consistency Analysis," *IEEE Transactions on Software Engineering*, vol, 24, no. 11, pp. 949-959, Nov. 1998.

[3] B. Smith, B. Millar, J. Dunphy, Y. Tung, P. Nayak, E. Gamble & M. Clark, "Validation and Verification of the Remote Agent for Spacecraft Autonomy," to appear in *Proceedings, 1999 IEEE Aerospace Conference.*

JPL

V&V of a Spacecraft s Autonomous Planner through Extended Automation

Martin S. Feather & Ben Smith
(Quality Assurance Office) (Information and Computing Technologies Research Section)
Jet Propulsion Laboratory
California Institute of Technology

Roadmap

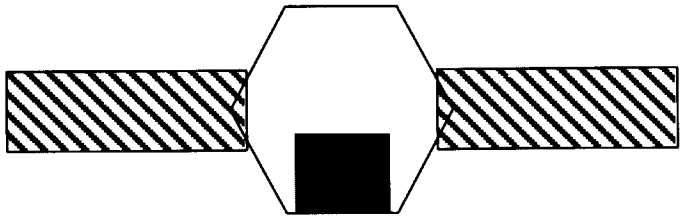
- Spacecraft's autonomous planner
- Testing challenges
- Example fragments
- Automate checking of flight rules
- Metrics
- Redundancy & Rationale
- Validation
- Applicability - worthwhile & viable
- Partnership development

Funded by NASA's Software Program, Office of Safety and Mission Assurance UPN #323-08-5b, and by the Autonomy Technology Program.

Spacecraft s autonomous planner

Autonomous - no human oversight or intervention

Planner has wide range of behaviors



Thrust off
Camera on
Take image
Take image
Camera off
Thrust on

"Fly by asteroid"

Detailed command sequences are generated by spacecraft's on-board planner

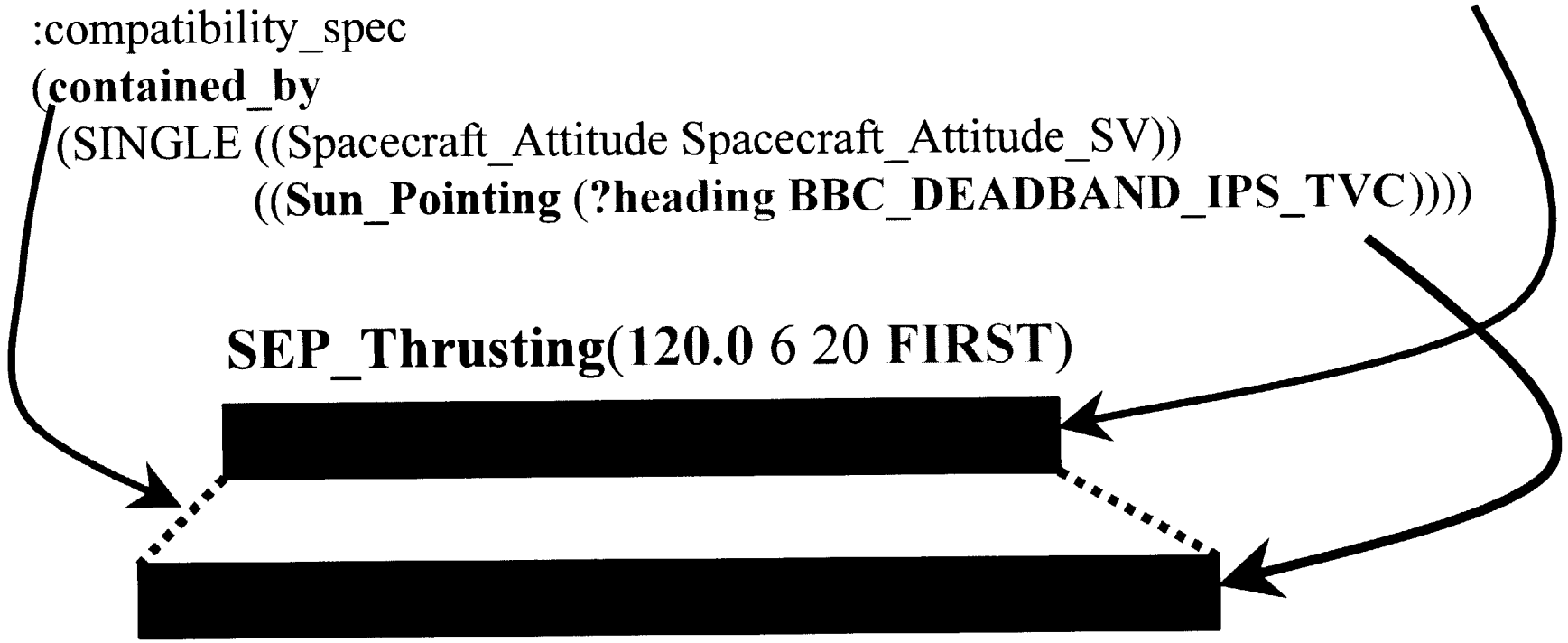
Some testing challenges

- Each plan must satisfy every one of the 200+ flight rules
  - each flight rule is a temporal relationship between activities
  - e.g., thrusting activity *contained-by* constant-pointing-on-sun
- Plans are detailed and voluminous (1,000 - 5,000 lines)
- Information dispersed throughout plan
- Thorough testing yields thousands of plans

Manual inspection impractical - need automation!

Example flight rule

```
(Define_Compatibility
        (SINGLE ((SEP SEP_SV))
                ((SEP_Thrusting (?heading ?level ?duration FIRST))))
 :compatibility_spec
 (contained_by
  (SINGLE ((Spacecraft_Attitude Spacecraft_Attitude_SV))
          ((Sun_Pointing (?heading BBC_DEADBAND_IPS_TVC))))
```



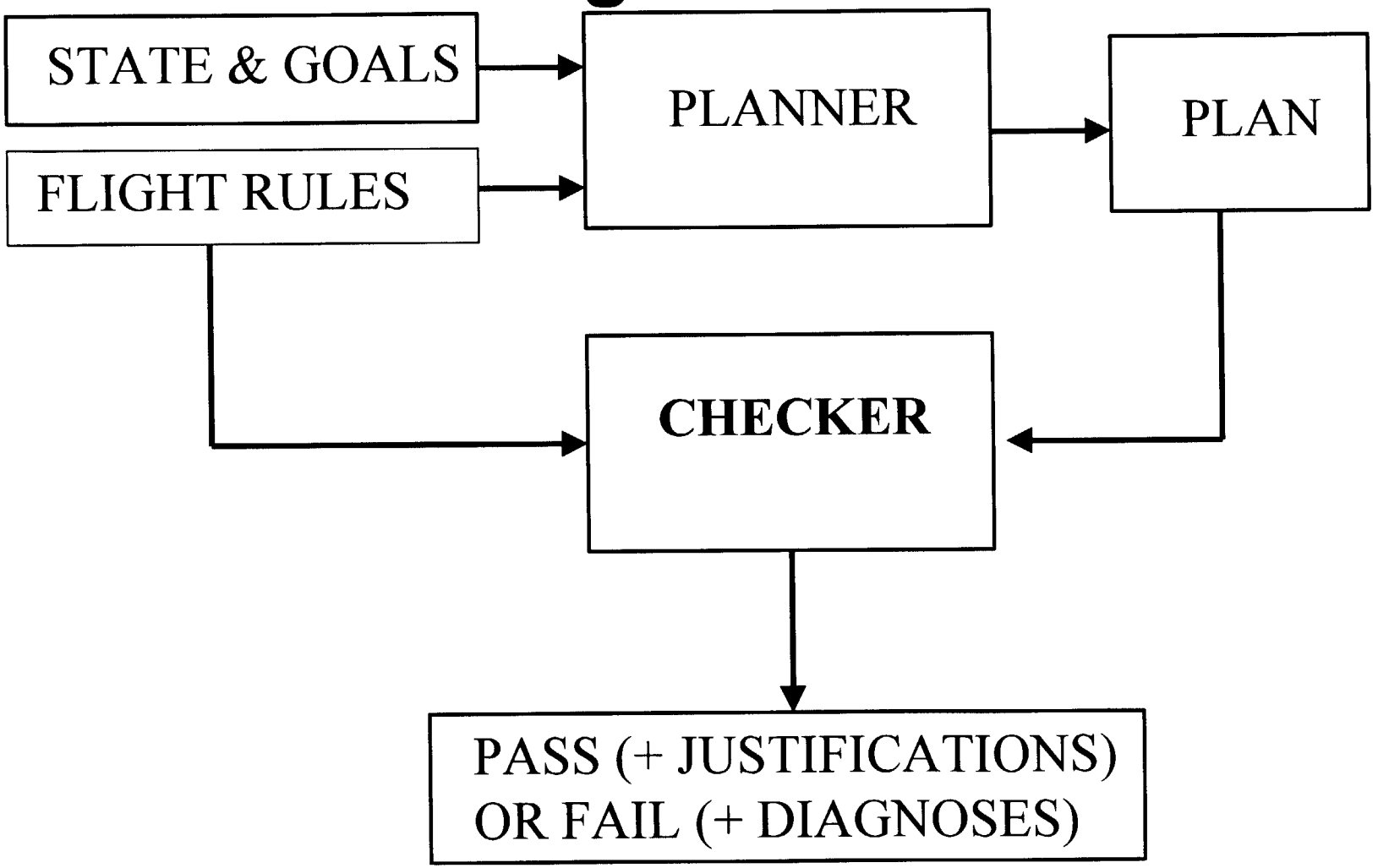
Rules involve time & parameters of states & activities

Small fragment of a plan

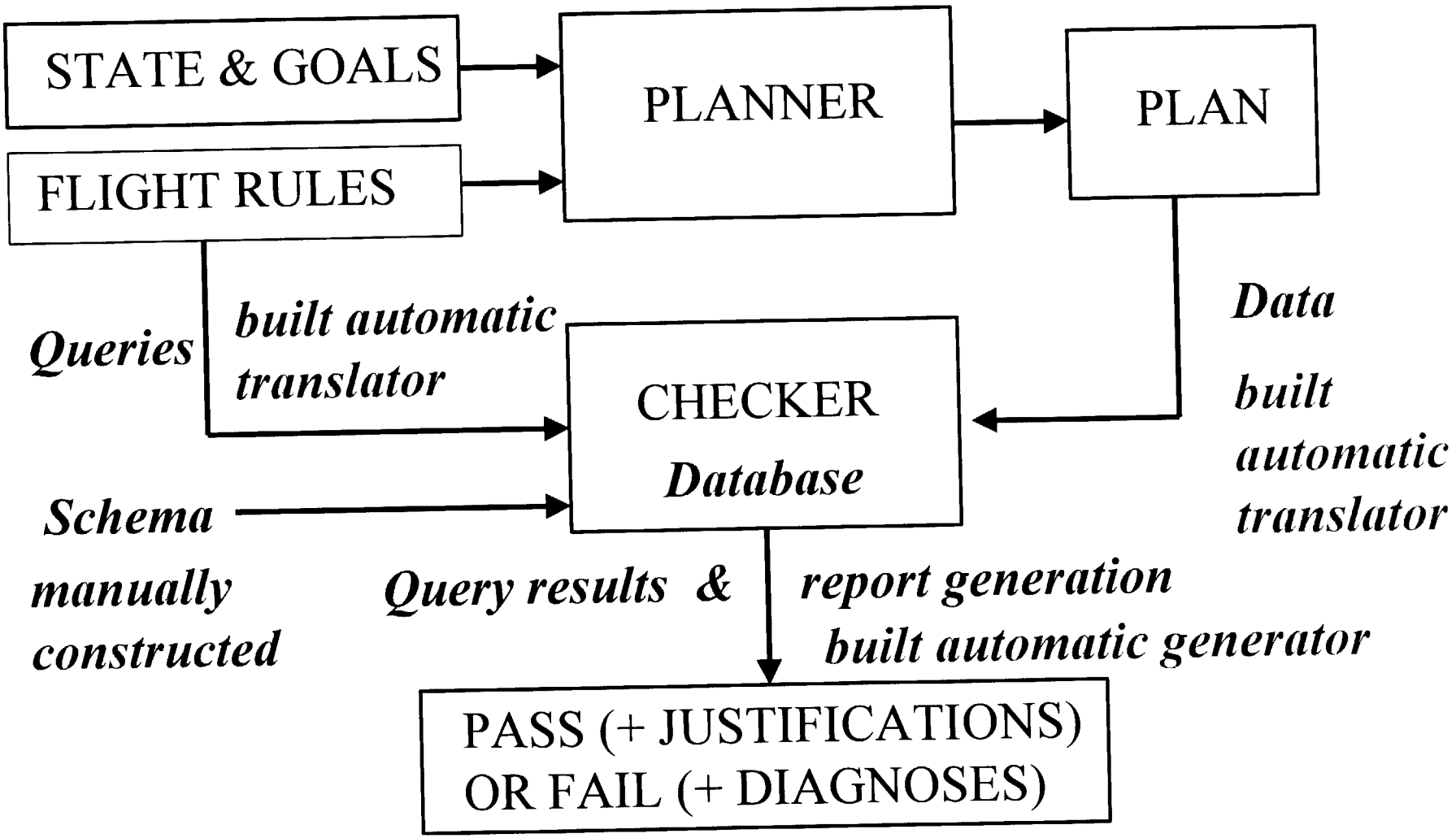
```
(#S(C-TOKEN
   :CARDINALITY :SINGLE    :NAME VAL-920
   :SV-SPEC (SPACECRAFT_ATTITUDE  SPACECRAFT_ATTITUDE_SV)
   :TYPE-SPEC ((CONSTANT_POINTING_ON_SUN
                  (HGA_AT_EARTH   BBC_DEADBAND_CRUISE)))
   :START-B-TOKEN VAL-920    :END-B-TOKEN VAL-920
   :STATE-VARIABLE (SPACECRAFT_ATTITUDE
                        SPACECRAFT_ATTITUDE_SV)
   :TOKEN-TYPE ((CONSTANT_POINTING_ON_SUN
                    (HGA_AT_EARTH    BBC_DEADBAND_CRUISE)))
   :DURATION (37801 500000000)
   :START-TIME-POINT TP-1279
   :END-TIME-POINT TP-1116
   :COMPAT-CONSTRAINTS ((CONTAINS 0 500000000 0 500000000)
                              PS_WAYPT_1))    )
```

Designed to be read by software, not by humans!

Automate checking plans against flight rules



Database used to perform checks



Metrics

- Automatically check every flight rule *> 200*
- Applied to plans generated during testing *thousands*
- Checking plans faster than generating plans
  30 seconds - 4 minutes *3 minutes - 10 minutes*
- Automatic regeneration of checker when flight rules change *< 10 minutes; done 3 times*
- Development of checker lesser effort than of planner
  months *years*
- Accommodated a change to plan syntax *< 3 days*

Redundancy & Rationale

A plan contains a sequence of activities and *justifications* of those activities -- *justification: activity* ←→ *flight rule(s)*

Rationale - planner arrives at the "right solution" (a plan that meets flight rules) for the "right reasons"

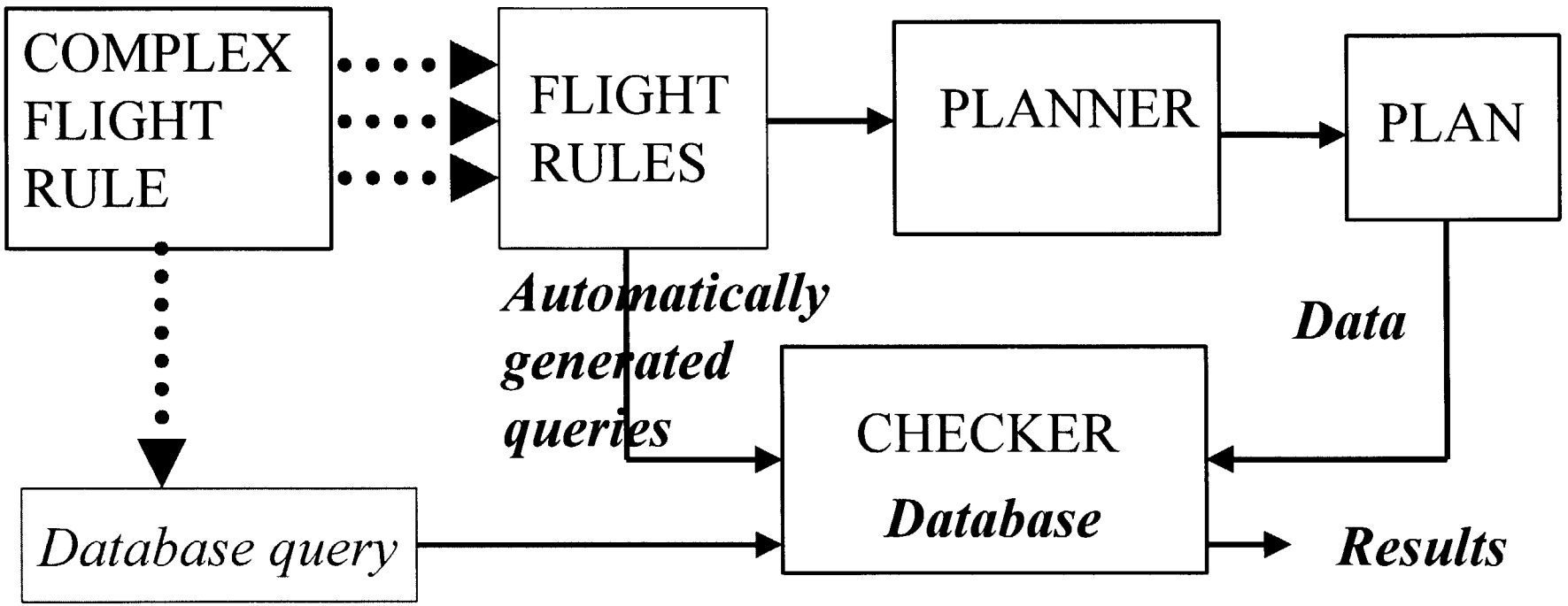
=> ***increased confidence in planner***

Redundancy - checker tests all the following:

- all activities of plan adhere to all flight rules
- each activity has a justification for every flight rule applicable to that activity
- every activity's justification can be traced back to an applicable flight rule

=> ***increased confidence in checker***

Validation



Complex flight rule *manually* decomposed and expressed as several planner flight rules.

Validate by manually expressing as a single database query.

Applicability - *worthwhile* when:

- Voluminous amounts of data to check
  - test run yields lots of data
  - lots of test runs
- Checking is difficult
  - many checks
  - checks hard to perform
- Cannot instead analyze the *generator* of the data
  - e.g., planner too complex for analysis via model checking

Applicability - *viable* when:

- Data self-contained w.r.t. check
- Data in machine-manipulable form
  - e.g., plan is input to another program on spacecraft
- Check in machine-manipulable form
  - e.g., flight rules expressed as planner constraints
- Checking easier than generation
  - e.g., planning - a core AI problem; checking easier
- Analysis language more expressive than requirements language
  - e.g., database language v.s. planner constraint language

Partnership development

Spacecraft planner expert - Ben Smith

Analysis expert - Martin Feather

- Spacecraft expert's time a critical resource
- Neither partner has time to become expert in both areas
- Analysis expert developed & maintained checker
- Spacecraft expert used checker
- Spacecraft expert extended checker (for validation)

Performing Verification and Validation in Reuse-Based Software Engineering

Edward A. Addy
NASA/WUV Software Research Laboratory

304-3678353 (Voice)
304-367-8211 (Fax)
eaddy@wvu.edu

1. INTRODUCTION

The implementation of reuse-based software engineering not only introduces new activities to the software development process, such as domain analysis and domain modeling, it also impacts other aspects of software engineering. Other areas of software engineering that are affected include Configuration Management, Testing, Quality Control, and Verification and Validation (V&V). Activities in each of these areas must be adapted to address the entire domain or product line rather than a specific application system. This paper discusses changes and enhancements to the V&V process, in order to adapt V&V to reuse-based software engineering.

V&V methods are used to increase the level of assurance of critical software, particularly that of safety-critical and mission-critical software. Software V&V is a systems engineering discipline that evaluates software in a systems context [Wallace and Fujii 1989]. The V&V methodology has been used in concert with various software development paradigms, but always in the context of developing a specific application system. However, the reuse-based software development process separates domain engineering from application engineering in order to develop generic reusable software components that are appropriate for use in multiple applications.

The earlier a problem is discovered in the development process, the less costly it is to correct the problem. To take advantage of this V&V begins verification within system application development at the concept or high-level requirements phase. However, a reuse-based software development process has tasks that are performed earlier, and possibly much earlier, than high-level requirements for a particular application system.

In order to bring the effectiveness of V&V to bear within a reuse-based software development process, V&V must be incorporated within the domain engineering process. Failure to incorporate V&V within domain engineering will result in higher development and maintenance costs due to losing the opportunity to discover problems in early stages of development and having to correct problems in multiple systems already in operation. Also, the same V&V activities will have to be performed for each application system having mission or safety-critical functions.

On the other hand, it is not possible for all V&V activities to be transferred into domain engineering, since verification extends to the installation and operation phases of development and validation is primarily performed using a developed system. This leads to the question of which existing (and/or new) V&V activities would be more effectively performed in domain engineering rather than in (or in addition to) application engineering. Related questions include

how to identify critical reusable components how to identify reusable components for which V&V at the domain level would be cost-effective, and how to determine the level to which V&V should be performed on the reusable components and on the domain architecture itself.

This paper discusses a framework for performing V&V within reuse-based software engineering that has been presented in [Addy 1998]. The framework identifies V&V tasks that could be performed in domain engineering., V&V tasks that could be performed in the transition from domain engineering to application engineering, and the impact of these tasks on application V&V activites. This paper further considers the extension of criticality analysis from an application-specific context to a product-line context.

2. DIFFERENCES BETWEEN V&V AND COMPONENT CERTIFICATION

Much work has been done in the area of component certification, which is also called evaluation, assessment, or qualification. These terms can have slightly different meanings, but refer in general to rating a reusable component against a specified set of criteria.

The common thread through these certification processes is the focus on the component rather than on the systems in which the component will eventually be (re)used. Dunn and Knight [1993] note that with the exception of the software industry itself, customers purchase systems and not components. Ensuring that components are well designed and reliable with respect to their specifications is necessary but not sufficient to show that the final system meets the needs of the user. Component evaluation is but one part of an overall V&V of an application system.

Another distinction between V&V and component certification is the scope of the artifacts that are considered. While component certification is primarily focused on the evaluation of reusable components (usually code-level components), V&V also considers the domain model and the generic architecture, along with the connections between domain artifacts and application system artifacts. Some level of component certification should be performed for all reusable components, but V&V is not always appropriate. V&V should be conducted at the level determined by an overall risk mitigation strategy.

3. FRAMEWORK FOR PERFORMING V&V W2ITHIN REUSE-BASED SOFTWARE ENGINEERING

One of the working groups at the 1996 Reuse workshop (Reuse '96) developed a framework for performing V&V within reuse-base software engineering [Addy 1996]. This framework is illustrated in Figure 1, and is described in more detail in [Addy 1998].

Domain-level V&V tasks are preformed to ensure that domain products fulfill the requirements established during earlier phases of domain engineering. Transition-level tasks provide assurance that an application artifact correctly implements the corresponding domain artifact. Traditional application-level V&V tasks ensure the application products fulfill the requirements established during previous application life-cycle phases.

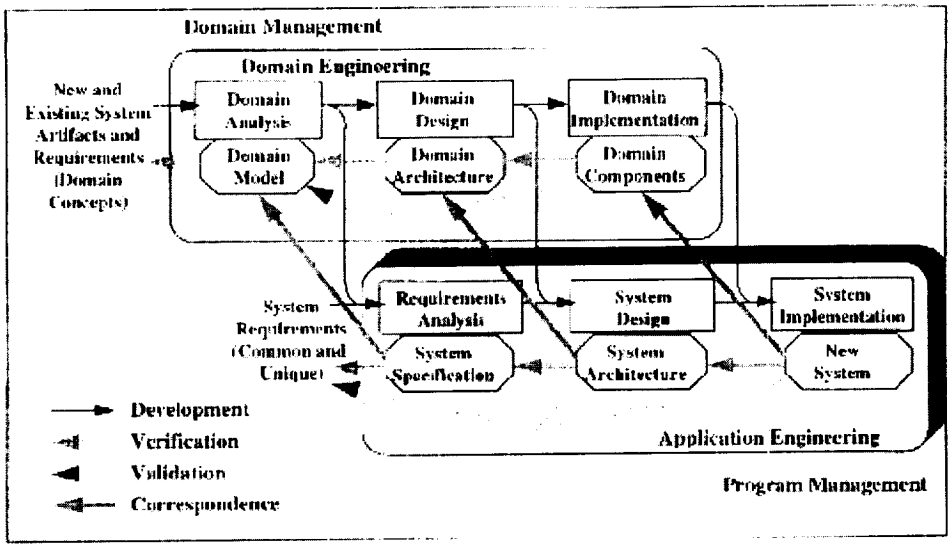


Figure 1: Framework for V&V within Reuse-Based Software Engineering

Performing V&V tasks at the domain and transition levels will not automatically eliminate any V&V tasks at the application level. However, it should be possible to reduce the level of effort for some application-level tasks. The reduction in effort could occur in a case where the application artifact is used in an unmodified form from the domain component, or where the application artifact is an instantiation of the domain component through parameter resolution or through generation.

Domain maintenance and evolution are handled in a manner similar to that described in the operations and maintenance phase of application-level V&V. Changes proposed to domain artifacts are assessed by V&V to determine the impact of the proposed correction or enhancement. If the assessment determines that the change will impact a critical area or function within the domain, appropriate V&V activities are repeated to assure the correct implementation of the change.

4. V&V OF GENERAL COMPONENTS

The discussion in Section 3 focused on the issue of V&V within domain engineering, in the situation where the final systems would be subject to V&V even if the systems were not developed within a reuse environment. Many of the same justifications for performing V&V in a product line that includes critical systems also apply to V&V of other product lines and to general purpose reusable components. The component Verification, Validation and Certification Working Group at WISR 8 found four general considerations that should be used in determining the level of V&V of reusable components [Edward and Wiede 1997]:

- Span of application – the number of components of systems that depend on the component
- Criticality – potential impact due to a fault in the component
- Marketability – degree to which a component would be more likely to be reused by a third party
- Lifetime – length of time that a component will be used

The domain architecture serves as the context for evaluating software components in a product-line environment. However, this architecture may not exist for general use components. The working Group determined that the concept of validation was different for a general use component than for a component developed for a specific system or product line. In the latter case validation refers to ensuring that the component meets the needs of the customer. A general use component has not one customer, but many customers, who are software developers rather then end-users. Hence validation of a general use component should involve the assurance (and supporting documentation) that the component satisfies a wide range of alternative usages, rather than the specific needs of a particular end-user.

5. CRITICALITY ANALYSIS

Although not shown as a specific V&V task for any particular phase of the life-cycle, criticality analysis is an integral part of V&V planning. Criticality analysis is performed in V&V of application development in order to allocate V&V resources to the most important (i.e., critical) areas of the software [IEEE Std 1059-1993]. This assessment of criticality and the ensuing determination of the level of intensity for V&V tasks are crucial also within reuse-based software engineering.

The Criticality Analysis and Risk Assessment (CARA) method is one of the tools used by V&V practitioners at the NASA IV&V Facility to evaluate and prioritize the areas of risk with an application system. The CARA method includes a determination of both the likelihood that an error will occur in the software (risk) and the impact of an error occurring in that software(criticality). A team determines the criteria by which the software will be evaluated, which includes areas such as performance and operation, safety, development cost, and development schedule. Each software component receives a score based on these factors, and the scores are performed once with each major development milestone.

Other methods that are used to perform criticality analysis (or risk analysis) include the Software Engineering Institute Software Risk Evaluation Method and the NASA Continuous Risk Management process based on the SEI SRE, the NASA Ames Research Center approach to risk management within an ISO 9001 environment, the Jet Propulsion Laboratory System Risk Balancing method, the V&V Goal-Question-Metric process, and the SAIC Risk Cube method. [WORM 98] All of these methods share with CARA a tabular-based aggregation approach to determine overall risk, by evaluating fundamental risk factors and combining them in some fashion to determine aggregate risk of a system, subsystem, or component.

Criticality analysis methods need to be extended to include consideration of multiple application systems developed from reusable components. Each software component should be evaluated not only as is done with current methods, but also with consideration of the criteria listed in Section 4. For example, a component that does not score highly on any specific application system might require a high level of V&V because it is used (or anticipated to be used) in a large number of systems.

The current methods to perform analysis on software architectures [Tracz 1996, Garlan 1995] are also directed at the architecture for an application system rather than a product line architecture. One of the approaches being researched is a scenario-based analysis approach, Software Architecture Analysis Method [Kazman, et al. 1996], that could be extended to product line architectures. In the area of correspondence tasks, the Centre for Requirements and

Foundations at Oxford is developing a tool (TOR) to support tracing dependencies among evolving objects [Goguen 1996].

6. CONCLUSION

The framework for performing V&V in traditional application system development can be extended to reuse-based software engineering. The extended framework allows the V&V effort to be amortized over the systems within the domain or product line. Just as with V&V in application systems development, V&V should be performed as part of an overall risk mitigation strategy within the domain or product line.

The primary motivation for V&V within domain engineering is to find and correct errors in the domain artifact in order to prevent the errors from being propagated to the application systems. This motivation is especially strong where the application systems perform critical functions. Even if there are no critical functions performed by the systems within the domain, V&V might be appropriate for a component that has the potential to be used in a large number of application systems.

ACKNOWLEDGEMENT

This work is funded by NASA Cooperative Agreement NCC 2-979 at the NASA/Ames IV&V Facility in Fairmont, WV.

REFERENCE

Addy, Edward A. (1998), "A Framework for Performing Verification and Validation in Reuse-Based Software Engineering," Annals of Software Engineering, Vol. 5, 199.

Addy, Edward A. (1996), V&V Within Reuse-Based Software Engineering," Proceedings for the Fifth Annual Workshop on Software Reuse Education and Training, Reuse '96', http//www.asset.com/WSRD/conferences/proceedings/results/addy/addy.html.

Dunn, Michael F. and John C. Knight (1993), Certification of Reusable Software Parts," Technical Report CS-93-41, University of Virginia, Charlottesville, BA

Edwards, Stephen H. and Bruce W. Wiede (1997), "WISR8L 8th Annual workshop on SW Reuse," Software Engineering Notes, 22, 5, 17-31.

Garlan, David (1995), "First International Workshop on Architectures for Software Systems Workshop Summary," Software Engineering Notes, 20, 3, 84-89.

Goguen, Joseph A. (1996), "Parameterized Programming and Software Architecture," In Proceedings of the Fourth International Conference on Software Reuse, IEEE Computer Society Press, Los Alamitos, CA, pp. 2-10.

IEEE STD 1059-1993, IEEE Guide for Software Verification and Validation Plans, Institute of Electrical and Electronics, Inc., New York, NY.

Kazman, Rick, Gregory Abowd, Len Bass, and Paul Clements (1996), "Scenario-Based Analysis of Software Architecture," IEEE Software, 13, 6, 47-55.

Tracz, Will (1996), "Test and analysis of Software Architectures," In Proceedings of the international Symposium on software Testing and Analysis (ISSTA '96), ACM press, New York, NY, pp 1-3.

Wallace, Dolores R. and Roger U. Fujii (1989), Software Verification and Validation: Its Role in Computer Assurance and Its Relationship with software Project Management Standards," NIST Special Publication 500-165, National Institute of Standards and Technology, Gaithersburg, MD.

WORM 98, Proceedings of the Workshop on Risk Management, October 1998.

Performing Verification and Validation in Reuse-Based Software Engineering

Software Engineering Workshop

Edward A. Addy

NASA/WVU Software Research Laboratory

NASA/Goddard Space Flight Center

Software Engineering Laboratory

December 3, 1998

Impact of Reuse-Based Software Engineering on the Software Process

- **Implementation of reuse-based software engineering (software product lines) impacts all areas of the software development process**

→Domain Engineering
→Component Development
→Reuse Library Maintenance

→Requirements Analysis
→Design and Implementation
→System Integration
→Testing

→Configuration Management
→Quality Assurance
→Verification and Validation

Framework for V&V in Reuse-Based Software Engineering

- Current V&V work geared toward specific application systems, throughout the lifecycle of the system

- Reuse-based software engineering (software product lines) introduces software engineering at a level outside the individual application system
  - →Common requirements
  - →Architecture for multiple systems
  - →Components used in multiple systems
  - →Unit testing on shared components
  - →Documentation

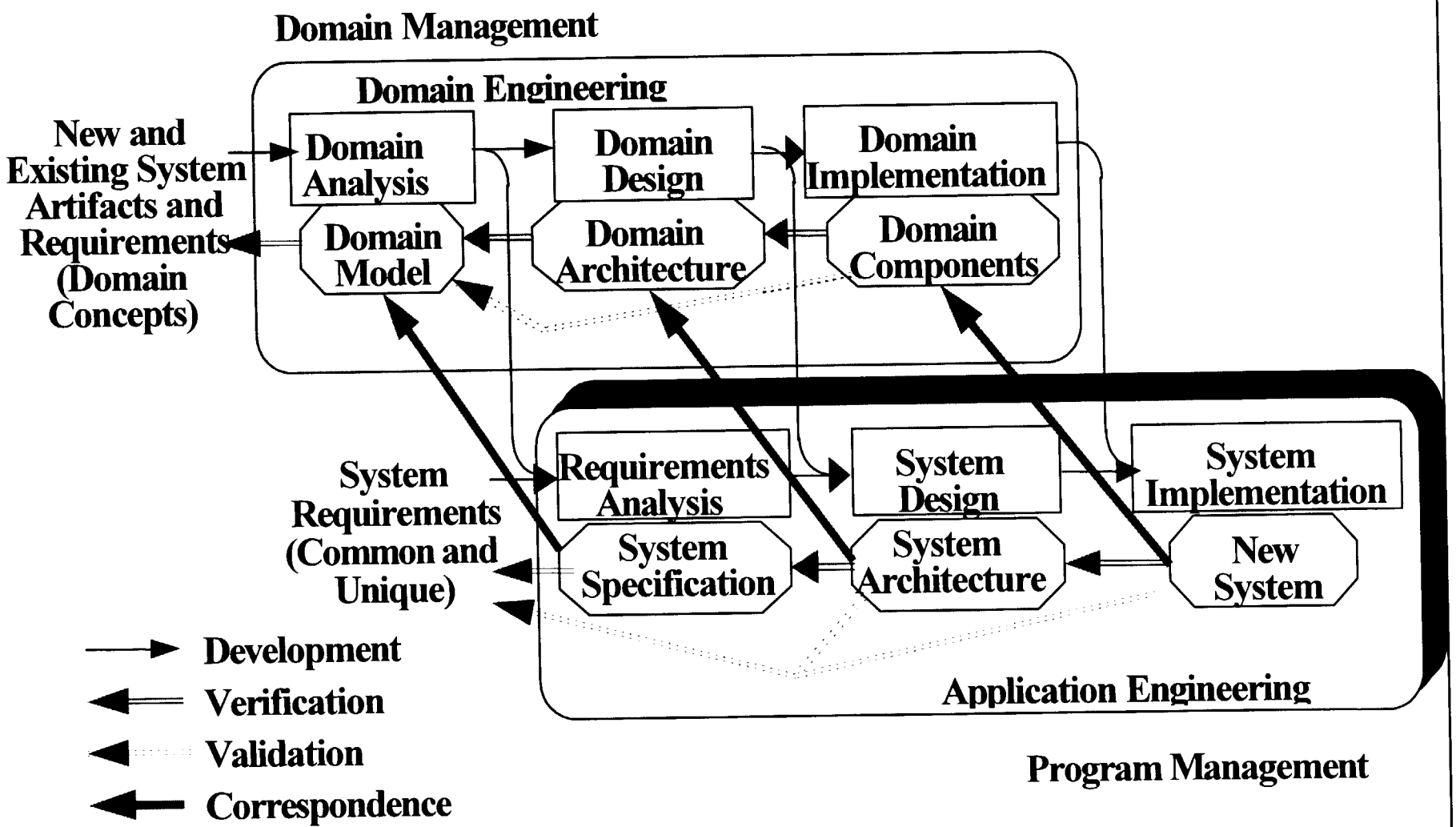
The Context for V&V

- **"Software V&V is a systems engineering discipline that evaluates software in a systems context"**

  Delores Wallace and Roger Fujii, Software Verification and Validation: Its Role in Computer Assurance and Its Relationship with Software Project Management Standards, NIST Special Publication 500-165

- **In reuse-based software engineering, the context for V&V is provided by the domain model and the generic architecture.**

General Framework



V&V Tasks

- **Domain-level: tasks performed to ensure that domain products fulfill the requirements established during earlier phases of domain engineering.**
- **Transition-level: tasks to provide assurance that an application artifact correctly implements the corresponding domain artifact.**
- **Application-level: traditional V&V tasks to ensure the application products fulfill the requirements established during previous application lifecycle phases.**

Domain-Level V&V Tasks

PHASE	TASKS
Domain Analysis	Validate Domain Model Model Evaluation Requirements Traceability Analysis
Domain Design	Verify Domain Architecture Design Traceability Analysis Design Evaluation Design Interface Analysis Component Test Plan Generation Component Test Design Generation
Domain Implementation	Verify and Validate Domain Components Component Traceability Analysis Component Evaluation Component Interface Analysis Component Documentation Evaluation Component Test Case Generation Component Test Procedure Generation Component Test Execution

Transition-Level V&V Tasks

PHASE	TASKS
Requirements	Correspondence Analysis between System Specification and Domain Model
Design	Correspondence Analysis between System Architecture and Domain Architecture
Implementation	Correspondence Analysis between System Implementation and Domain Components

- Map the application artifact to the corresponding domain artifact.
- Ensure that the application artifact has not been modified from the domain artifact without proper documentation.
- Ensure that the application artifact is a correct instantiation of the domain artifact.
- Obtain information on testing and analysis on a domain artifact to aid in V&V planning for the application artifact.

Criticality Analysis in Reuse-Based Software Engineering

- Criticality analysis is performed in order to allocate V&V resources to the most important (i.e., critical) areas of the software
- Question - How should criticality analysis be extended from V&V in application software engineering to V&V in reuse-based software engineering?
  - →Component more critical in one system than in another
  - →Component not critical in any individual system, but used in many systems (and hence critical to the organization)
  - →Component will be replaced quickly in most systems, but will remain for a long period in a few systems

Economic Considerations for Reusable Components

- Span of application – the number of components or systems that depend on the component
- Criticality – potential impact due to a fault in the component
- Marketability – degree to which a component would be more likely to be reused by a third party
- Lifetime – length of time that a component will be used

Source - Working Group on Component Verification, Validation, and Certification, WISR8, March 1997

Session 6: Embedded Systems and Safety Critical Systems

Defining and Validating Embedded Computer Software Requirements Using the ECS, OTPM and IPFA
J. Manley, University of Pittsburg

Using Automatic Code Generation In the Attitude Control Flight Software Engineering Process
D. McComas, J. O'Donnell, Jr., and S. Andrews, NASA/Goddard

Determining Software (Safety) Levels for Safety Critical Systems
M. Yin and D. Tamanaha, Raytheon Systems Company

Defining and Validating Embedded Computer Software Requirements

Using the ECS, OTPM and IPFA

Dr. John H. Manley
Professor of Industrial Engineering
Director, Manufacturing Systems Engineering Program
University of Pittsburgh
jmanley@engrng.pitt.edu

1 Introduction

In the 1960s a new class of highly specialized digital computers began to evolve from the existing worlds of general purpose "automatic data processing" business machines and specialized analog computers. In 1973 this new computer type was first formally defined by the author for the US Air Force as an "embedded computer" since they were being engineered into so called "embedded computer systems" (ECS).[1] An ECS was defined as a stand-alone, real-time, semi-automated system such as the then-new B-1A strategic bomber aircraft. Since the early 1970s embedded computers have become ubiquitous and appear as integral parts of most manufactured products, from talking toys and automobiles to aerospace vehicles. They are also used extensively to integrate complex repetitive processes which represent the nervous systems of large-scale enterprises such as satellite launch centers and modern factories.[2]

This paper describes a field-tested, common-sense, macro-modeling solution to the now 25-year-old problem of how to cost-effectively define and validate certain critical embedded computer software requirements. The solution employs the author's Object Transformation Process Model (OTPM) [Figure 2] which can be used to macro-model any ECS architecture and comprehensively identify the minimum essential information (MEI) required by specified humans, machines, and computers involved in time-critical processes.[3] OTPM-based modeling results in: (1) the identification of explicit inputs and outputs required for each involved embedded computer program, and (2) the required timing for data arrival at and computer information departure from any embedded computer. This information is used to develop an improved requirement definition for the stored embedded computer programs needed to transform embedded computer input data sets into required output computer information to humans, control signals to machines, or computer data to other embedded computers. Finally, output timing and data volume specifications can be derived from this information and used to define bandwidth requirements for designing ECS supporting telecommunication systems.

2 ECS Computer Information Flow Analysis

All automated manufacturing lines, manned space vehicles, military aircraft, nuclear power plants, and similar real-time automated systems contain three control elements: (1) embedded computers, (2) humans-in-the-loop, and (3) certain electromechanical and/or analog devices, e.g., switches, sensors, and motors. The three elements are interconnected to make up the parent system and both individually and collectively require precisely-timed, highly-accurate, closed-loop control systems to ensure both human safety and high quality output products and/or services from their parent systems. Digital computers embedded in time- and safety-critical systems are especially difficult control elements to initially design, validate, and subsequently upgrade during parent systems reengineering projects, e.g., modernizing factories, and finding sources of year 2000 (Y2K) problem code in embedded processors.

Industrial engineers can assist computer systems and communication engineers design new and/or reengineer troublesome or obsolescent real-time ECS by ensuring that all control loops, both feed forward and feed back, are complete and efficient. The primary tool used for such work is a modified industrial engineering Process Flow Analysis (PFA) procedure called an Information Process Flow Analysis (IPFA).[4] The IPFA uses a combination of an ECS physical model [Figure 1] and the OTPM [Figure 2] as a conceptual framework to guide specialized analyses of complex, large-scale, embedded systems. When the IPFA is used to reengineer systems that are considered "troublesome" by management, the analysis objectives are twofold: (1) identify the minimum essential, necessary and sufficient information required for controlling any type of digital or analog computer embedded in large, complex, real-time automated systems, and (2) eliminate unnecessary and/or redundant information to and from embedded computers in order to improve the parent system's overall throughput and efficiency. What is different about IFPA is that it is focused on improving information flows—not material flows. Thus, the normal use of PFA by industrial engineers to eliminate waste from moving, storing, and unnecessary handling of physical material is of secondary importance to IPFA. The primary focus of IFPA is on ensuring that all control loops, both feed forward and feed back are complete, validated, and made as efficient as economically possible.

2.1 Embedded Computer System (ECS) Model Description

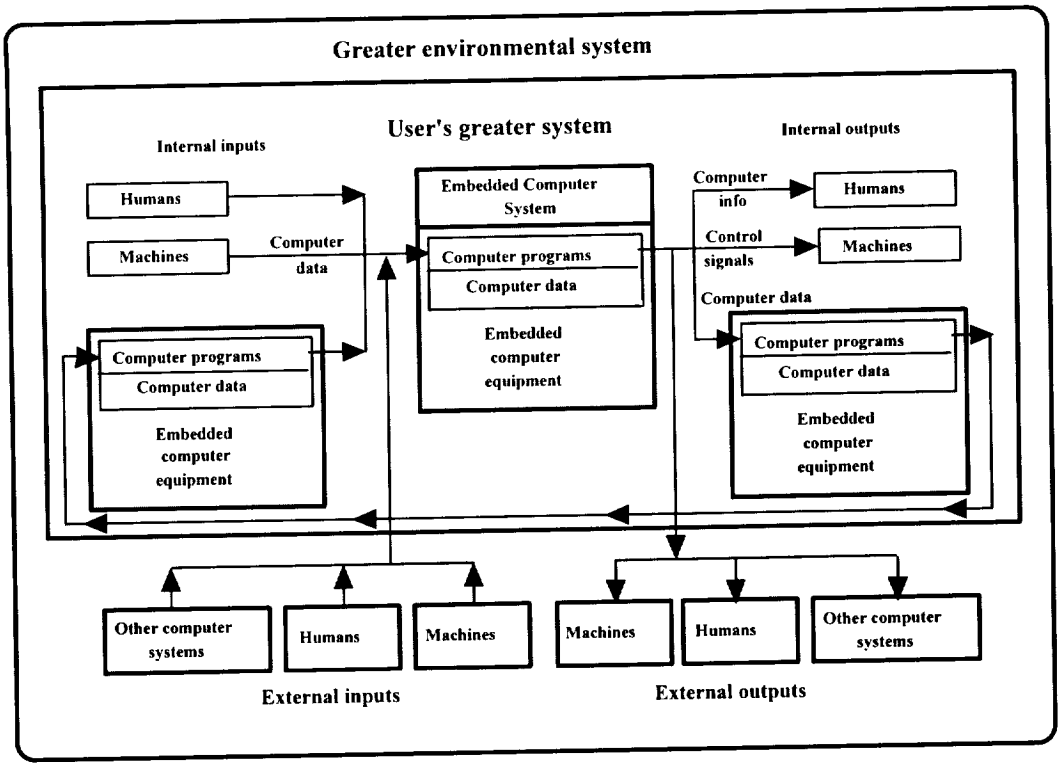


Figure 1: Embedded Computer System Model

The ECS model in Figure 1 has the following four important attributes:

1. The ECS describes *systems nested within systems*. An embedded computer system can be a single processor containing computer software (computer programs and computer data) which controls an electromechanical system, such as the flight controls of an aerospace vehicle, or desired pacing of an automated materials handling system. A "user's greater system" is composed of an integrated complex of humans, machines and multiple embedded computer systems that comprise, for example, a highly-automated aircraft, or an entire factory. A "greater environmental system," such as an airport that contains a control tower, aircraft maintenance facilities, etc. provides external inputs and receives outputs to support several "greater user systems" such as a fleet of aircraft. This nesting approach, or "onion skin model" can be continued upward to higher conceptual levels, such as multiple airports connected together by an air traffic control system, and so on.

2. The ECS emphasizes *three distinct sources of ECS input computer data*: (1) data generated by humans such as a single digital pulse from pushing a button, to a complex keyed or voice input data string, (2) analog or digital data generated by machines, such as a position signal from a servomechanism or the operation of a limit switch, and (3) digital data generated as output from another ECS within a user's greater system.

3. The ECS emphasizes *three distinct types of ECS outputs*: (1) computer-generated (human-understandable) information for humans, (2) computer-generated (machine-understandable) control signals for electromechanical machinery, and (3) computer-generated (computer program-understandable) output computer data from one ECS's computer program for another ECS's computer program as input data.

4. The ECS emphasizes that *all types of ECS inputs and outputs can coexist within a user's greater system, and also can originate from and terminate in the ECS's greater environmental system.*

Communication system issues addressed in the context of the ECS involve the physical and logical interconnections between embedded computers, humans, and machines that are integral to enterprise-wide manufacturing information systems. Today's manufacturing and aerospace systems engineers must not only understand the fundamentals of communication technology as it applies to embedded computer systems, but they must also know how to intelligently use this capability to install successful factory, office, and manned vehicle automation systems.

2.2 Object Transformation Process Model (OTPM) Description

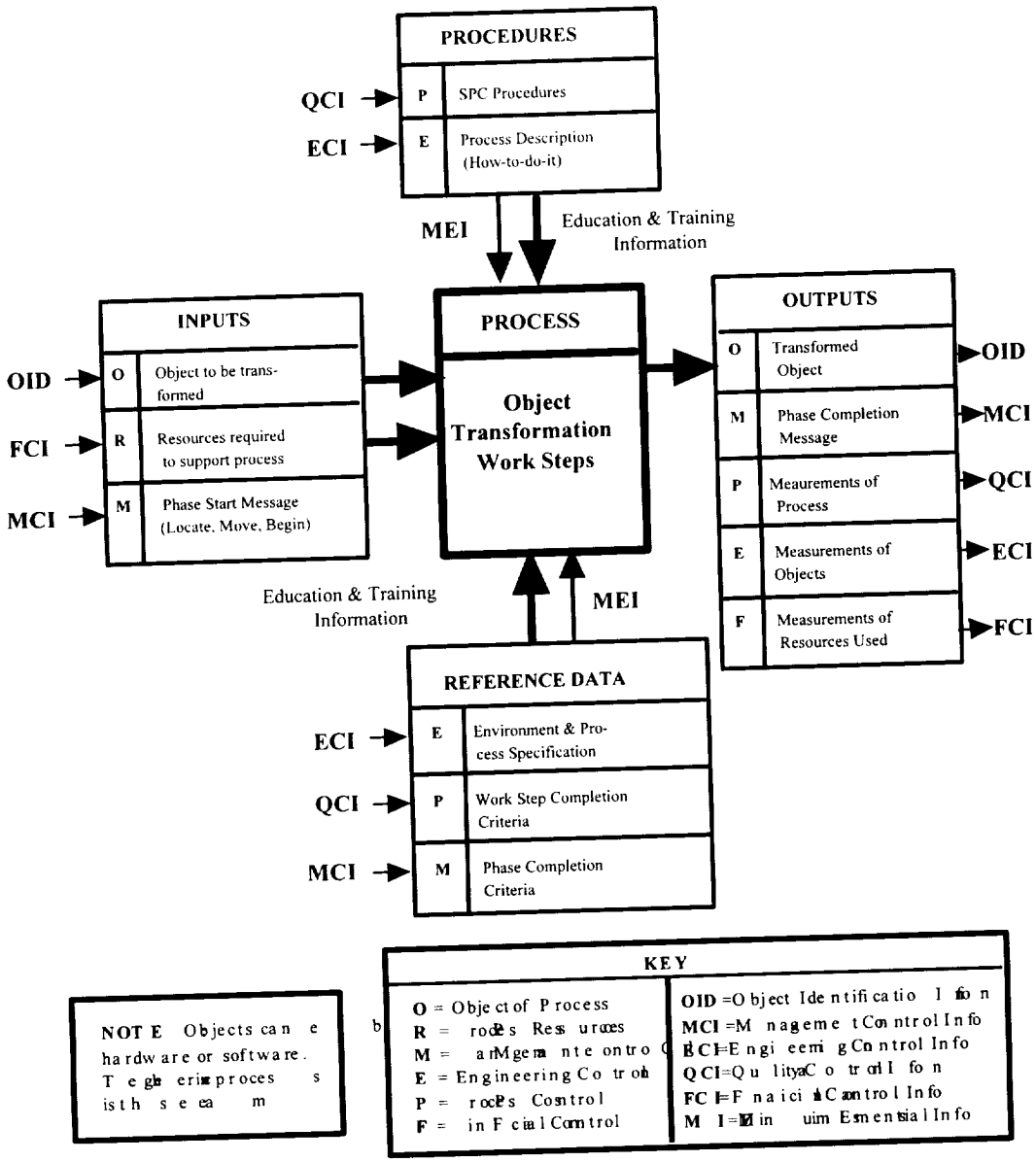


Figure 2: Object Transformation Process Model (OTPM)

The OTPM in Figure 2 describes a generic, arbitrarily-defined object transformation process (OTP) phase (which can be defined at any level of detail) and four information support components: inputs, outputs, procedures, and reference data. The physical input object is the output of a preceding transformation process phase. In the case of manufacturing, the starting point of a birth-to-death, repetitive manufacturing life cycle process considers the first objects to be transformed are crude oil or metal ore out of the ground; or biological materials from the surface such as tomatoes or peaches for transformation into food products. This first phase of the manufacturing process is defined as refining which produces raw materials or producer goods such as fuels, plastic, wood, metal, or specialized food items such as pastes or purees. In OTPM terms, refining represents the first phase of the manufacturing OTP life cycle. Therefore, the object transformation sequence from birth to death is as follows:

- Refining or raw material processing.
- Raw material production.
- Parts fabrication.
- Product assembly.
- Product modification, repair and/or re-manufacturing.

- Product disassembly for recycling and/or reuse.
- Product destruction and/or disposal.

The input object to each phase is, of course, any object that was transformed by a previous process step. In the case of intelligent product manufacturing, e.g., anti-lock braking systems for automobiles, all software-intensive component parts are treated in the same manner as hardware objects using the OTPM since all software eventually ends up embedded in a physical component, i.e., logic chips, ROM, "firmware," etc., or "modules" in the case of autos. Thus, in addition to a physical material transformation process, there also exists an *intellectual object transformation process* that, for example, transforms in a stepwise manner a human's mental concept of a computer program (software) into a physical read-only-memory (ROM) configuration that is eventually assembled into some "smart" product. Hence, the OTPM was designed to satisfy this manufacturing systems engineering need for using the same generic model to describe both physical and "intellectual object" transformations, namely hardware and software. The OTPM requires any manufacturing process phase or work package to contain four basic categories of information, each of which is an integral part of the OTPM. As indicated above, one relates to object identification, and three to process control.

- OID—Object identification information uniquely describes the object of interest. This information accompanies the object and changes as the object is progressively transformed.
- MCI—Management control information denotes both specific direction from transformation phase supervisors as input and reference data, as well as feedback output information that returns to the supervisors.
- ECI—Engineering control information comprises "how-to-do-it" process descriptions and process environment specifications as input, and transformed object measurements as feedback information to the engineers.
- QCI—Quality control information comprises work step completion criteria and statistical process control procedures as input information, and measurements of the transformation process as feedback output information to the quality control personnel.

2.3 Engineering Control Information Perspective

Traditional manufacturing engineering control connotes the process of ensuring that a transformed object meets or exceeds tolerances specified by an engineering design drawing. The OTPM uses the term "engineering" in a much broader sense. For example, if the object transformation process involves preparing an annual corporate tax return, the "engineer" who defines the how-to-do-it procedures in all likelihood would be a certified public accountant (CPA) trained in tax matters. The point is that the transformation process must be specified by an *expert* who, upon receiving detailed feedback information on what the transformed object looks like (a completed tax return), can readily identify errors and make rapid corrections to the methods and tools being used (machines or software) that perform the transformation. For specifying parameter settings on complex machine tools, this is clearly the responsibility of, e.g., mechanical, electrical, manufacturing or software engineers.

2.4 Quality Control Information Perspective

The term "quality" has many different meanings, especially in the context of a manufacturing enterprise. For OTPM purposes, quality is a primary attribute of not only every object that flows through an enterprise, but also the process itself. In particular, process stability (the ability to repetitively produce identical engineering results) is of major concern, as is the identification of object defects that cannot readily be controlled by engineering designs, e.g., those that result from human errors. Therefore, for any given level of quality of an object input into a subsequent transformation process phase or work step, the transformation process itself will determine the output level of object quality which has two components: (1) that which engineering can control, and (2) that which has to do with efficiency. For example, in one manufacturing plant, a particular fabricated part may continually conform to engineering specifications but takes twice as long to produce in another plant. The difference can be attributed to *process variability* (efficiency) since object variation is not an issue.

The author has arbitrarily decided to focus a highly desirable "continuous process improvement" function under the umbrella of quality control to distinguish it from traditional engineering quality control. The OTPM model also removes quality control from consideration as a testing and assurance function, in keeping with a very important principle that quality be built into products to prevent defects, i.e., "do it right the first time."

Therefore, from a manufacturing information systems perspective, quality control information (QCI) is focused on making continuous process improvements in regard to both shortening process cycle times, and also reducing process variability by identifying new engineering and management controllable variables for incorporation into the engineering and management control systems. QCI is collected for all important object transformation processes

using real-time statistical process control (SPC) techniques. QCI is fed back to a new paradigm quality control organization for appropriate analysis and action as part of enterprise-wide "total quality" initiatives.

3 Enterprise-wide Process Integration Through the OTPM

As shown in Figure 2, the OTPM elaborates each of the input, output, process, and control boxes by labeling information flows for objects [O], management [M], engineering [E], financial [F], and process (quality) control [P]. Resources required for a specific transformation as part of financial budget control is designated [R]. From a business perspective, no object transformation process can be undertaken without direction from management, since it represents a direct expenditure of material, human, and other resources. Also, no transformation can be considered complete unless information is generated and fed back to the manager who authorized the operation. Finally, for every controlled process, management must provide an explicit criteria for successful completion.

After completing their planning and organizing functions, managers direct manufacturing operations through an input (phase start message). They also control through the output (phase completion message) and reference data (phase completion criteria). By time-stamping input and output messages to and from any process, managers (or engineers) can determine any process end-to-end cycle time. Conformance to the reference criteria and procedures helps control the quality of each operation. For both human and machine processes, effective control is established because people (and machines) are expected to "perform as they are measured."

The sheer simplicity of this model can be deceiving, since it can also integrate a number of processes within a group, department, division, or even an entire manufacturing plant into a single process. This can be accomplished by connecting outputs (feedback messages) from one object transformation process as the input to another process (direction message). To satisfy these integration requirements, we can theoretically combine the individual process models of an entire enterprise using their information flows as depicted by the OTPM using the same linking method. The main feature of this architectural building block is its *uniform four-dimensional interface for any process down to individual work steps*. Thus, complex *logical* process models having this configuration can be linked together in essentially the same manner that children build complex static structures out of Legos, and systems engineers design and construct dynamic worldwide telecommunication systems. The key to success in both cases are straightforward and understandable logical interfaces between the component parts. From the resulting logical models, we can construct viable *physical models* for controlling any business, engineering, or other control system. This is done using the ECS model referenced above in a similar manner to how it has been used by systems engineers for almost two decades to architect complex aerospace systems.

3.1 IPFA Link to Information System Requirements Specifications

Initial information requirements (or subsequent deficiencies) identified and documented through the IPFA must be translated into requirement specifications for engineering or reengineering supporting throughput, financial, business, engineering, and or total quality information systems in accordance with the OTPM conceptual framework. Note that the Structured Analysis and Design Technique (SADT) tool[5] that has been enhanced by the Air Force into IDEF (the Integrated Computer and Manufacturing ICAM DEFinition methodology)[6] appears on the surface to be similar to the OTPM IPFA methodology in regard to reengineering information systems that control real-time embedded manufacturing systems. However, the OTPM provides an added cross-functional higher level perspective of an enterprise that is not normally described by IDEF. In accordance with the IPFA methodology, traditional PFA provides $IDEF_0$ with task information, follow-up OTPM analysis provides $IDEF_1$ with what information is required, and $IDEF_2$ adds the missing element of when the information is required, as can be conceptualized from the ECS model.

Note also that IDEF, OTPM and IPFA are based on classical structured techniques, and are therefore compatible with information system software requirement specification development tools such as data flow diagrams, structure charts, entity-relationship diagrams, and most Computer Aided Software Engineering (CASE) tools. Other potential candidates for IPFA linkage also exist such as a state-based methodology using Requirements State Machine Language (RSML) which involves a graphical specification language that is both readable and reviewable by applications experts who are not computer scientists or mathematicians.[7] Note, however, that neither IDEF or RSML are *object-oriented* in the sense of emerging software development methodologies.[8] Therefore, one important subject of my university research is to find even more effective ways to link IPFA structured analysis results to object-oriented and/or other types of information system software requirement specification methods and tools.

In short, the OTPM and ECS conceptual models provide a framework to guide process flow analyses that can help industrial and manufacturing system engineers identify critical information errors of commission and omission during product and service manufacturing system reengineering projects. In addition, traditional industrial

engineering PFA is enhanced by adding the requirement to analyze the necessity and sufficiency of object identification and process control information that supports physical and/or intellectual object transformation processes. Finally, IPFA results can be used to develop requirement specifications for system engineering or reengineering any type of product or service manufacturing information system.

4 Using the OTPM to Improve Designer, User and Constructor Mutual Understanding

The totality of OTPM analysis and modeling methodology outputs specify multifaceted and integrated real-time information systems that support end-to-end physical and/or intellectual object transformation processes, primarily products and/or services that satisfy customer's expressed requirements. According to Olle,[9] any design product resulting from the design activities should include specifications that are understandable to the acceptors. For example, these may be the set of *user acceptors* who are required to review and submit positive approval of the design work. The specifications may be also reviewed by a *constructor acceptor*, who will apply very different judgments from the user acceptor.

Since the OTPM methodology focuses almost exclusively on the user, achieving mutual understanding between OTPM designers and users should not pose any major problems. However, special attention must be paid to establishing effective communications between the OTPM information system engineer and the constructor acceptors, the latter being primarily information system hardware, software and communications professionals. One of the biggest barriers to good communication in this case is the highly specialized technical jargon that each party uses to communicate with each other—which is a continuing problem for information system professionals throughout the world. In this case, the solution to achieving mutual understanding is to prescribe an *interface language* made up of carefully chosen and defined words and phrases which can be mutually understood. Unfortunately, to the best of my knowledge there are no universal interface language standards to enhance mutual understanding among information system analysts, designers and constructors. Therefore, the concepts and terminology recommended by the IFIP are used to the maximum extent possible in the OTPM methodology specification to help make OTPM-based design products more easily understood by constructor acceptors. In this regard, key elements of the OTPM methodology are summarized in the tables below which compares it to some of the traditional methods mentioned above. Additional details can be found in the author's book, *Rise Above the Rest: The Power of Superior Information, Knowledge, and Wisdom.*[10]

Process Orientation

OTPM	TRADITIONAL
End-to-end requirement-to-delivery product and service repetitive processes (manufacturing)	Data flows and their supporting processes
Cross-functional processes are required	Processes normally limited to selected functional areas
Enterprise-wide process integration promoted through the OTPM methodology	Process integration discouraged by traditional methodology (becomes too complex)
Optimize end-to-end process information requirements	Suboptimize process functions without regard for precedent or subsequent functions

Data Orientation

OTPM	TRADITIONAL
Minimum essential information (MEI) to support object transformations	All available information and relationships between data elements NOTE: This is especially troublesome with current enterprise requirements planning (ERP) methods
Data are categorized into management, financial, engineering, quality, and object identification	No standard method for categorizing data
Data are defined from a process definition	Data are cross-referenced to functional areas to ensure availability

Behavior Orientation

OTPM	TRADITIONAL
Transformation process corrective actions taken by humans, machines, or computers are triggered by means of comparisons of input with output MEI	Computerized tasks are triggered by human, computer or machine actions
OTPM specifies where and when MEI are needed to support human-to-human mutual understanding	Specifies user system events which trigger changes in data flow processes

Modeling Technique

OTPM	TRADITIONAL
Information Process Flow Analysis (IPFA) used to identify and model relationships of MEI needed to support any "manufacturing" process[11]	Required data elements identified by user(s) and information system analysts model their relationships
Embedded Computer System model used as a common framework for physical implementation of OTPM enterprise-wide architectures	No common interface model to link logical data flow models to physical architectural designs
MEI defined for intellectual and material object transformation process phases help determine physical location and type of databases required	Existing or planned database systems determine their physical location

5 OTPM Solution Field Experience and Results

Manufacturing industry sponsored field tests beginning in 1991 have clearly demonstrated that the OTPM and IPFA are highly useful for improving data identification, collection, analysis, feedback, and related manufacturing processes in ECS environments, especially for information system improvement. Most experiments have been conducted as integral parts of University of Pittsburgh Manufacturing Systems Engineering Program internships which required students to complete either a major capstone research project or formal M.S. thesis internship with their committees composed of both faculty and industry project supervisors.

1. The first major field test was conducted at the Packard Electric Division of General Motors Corporation in Warren, Ohio where an OTPM-modified PFA of a 320-machine plastic molding plant was used to identify and eliminate waste generated by ineffective communications in both business and engineering shop-floor semi-automated control systems. The improved information systems produced thesis-documented savings of over $1,800,000 per year.

2. The second major field test was conducted at AEG Westinghouse Transportation Services Inc. [currently ABB Adtranz] in West Mifflin, Pennsylvania that focused on a cycle time analysis of airport people mover manufacturing final assembly and testing. The OTPM-modified cycle time analysis uncovered the need for improved engineering control data. When such data were subsequently defined, collected and analyzed, the results led to making major changes to the final assembly process. This, in turn, generated an annual savings of over $600,000 and doubled the manufacturing plant's overall production capacity

3. Another early field test was conducted at Bloom Engineering, Inc. that manufactures customized metallurgical furnace control systems, each of which is uniquely designed for a single customer need. The focus there was on improving the generation and control of product design information to identify and separate that which is unique to a job from that which can be reused on future jobs. A new engineering OTPM-based database was designed and programmed which significantly improved Bloom's front-end engineering design processes.

Since 1993, 80 practicing engineers who average 11 years of industrial experience have learned how to implement the methodology described in this paper by taking the author's graduate course in manufacturing information systems reengineering. Most importantly, they have carried out 23 team projects to identify and solve ECS-based information system problems using the OTPM-based methodology in a variety of manufacturing and service organizations. In every project to date, the student teams were able to make significant improvements to existing automated processes, or plan information system solutions which were accepted by the target enterprises for subsequent implementation.

In addition to university research projects, the author has been teaching the methodology to other industrial organizations through short course offerings and also as part of his corporation's educational services offerings to government and industry. A sample of the wide variety of information system problems that have been solved by practicing engineers (as MSEP student researchers) using the OTPM methodology are provided in the table below.

ORGANIZATION STUDENT RESEARCHER	INFORMATION PROBLEM SOLVED
Petroleos de Venezuela (PDVSA) Hugo R. Vasquez-Tarbay	The OTPM was used to analyze the global RDC-generation process in Venezuela's nationalized oil company from the arrival of an RDC [purchase requisition in Spanish] at PDVSA Services, Inc. in Houston, Texas [PDVSA central purchasing agency in the US] to the selection of the panel of vendors for quotation. A relational database prototype was developed to unify 70 different procedures for RDC generation being used to gather information from PDVSA's country-wide operating divisions. This solution radically improved the quality of the selection of the computer codes by clients and subsidiaries.
Delphi Packard Electric Systems, General Motors Corporation Annette M. Pohlman	Applied the OTPM-based enterprise continuous improvement strategy to develop a measurement system for streamlining General Motors Corporation's multi-plant automobile power and signal distribution system product development process. The strategy was adopted for implementation by management.
Cornelius Architectural Products, Inc. Mark F. Rothert	Designed a standard coding system for communicating complex information relative to project material and components within a manufacturing organization to eliminate waste from interpersonal misunderstandings. The projects by Cornelius were one-of-a-kind, e.g., creating complex signage displays for Disneyland in Orlando, Florida which were exceedingly difficult to describe in understandable detail at the front end of the manufacturing process.
Superior Valve Company Tracy R. Shaffer	A total process perspective was used to define a model for assembly manufacturing cell design that incorporated product demand. Specifically, the model showed that once machine-component groupings are formed, the cell formation process can be further refined by sorting the product family by demand and grouping parts into replicated cells according to high and low to medium volume parts. The layout for each replicated cell was matched to the grouping's production demand and adequate cell utilization is now maintained by comparing demand to capacity for each replicated cell.
Delphi Packard Electric Systems, General Motors Corporation Daniel D. Gottfried	Used OTPM information process flow analysis (IPFA) to help define the minimum, relevant information systems requirements of a five-stage manufacturing process for producing automotive ignition cable. The problem solved was determining the root cause and implementing corrective actions to eliminate loose core cable defects from the extrusion of composite, high temperature core, rubber insulated ignition cable. "The use of object-oriented information systems facilitated more concise problem definition and timely discovery of causal factors that could be controlled or eliminated within the improved manufacturing system."
The Elliott Company Elizabeth A. Samstag	Used OTPM principles to analyze and improve the receiving function for a gas turbine power generator manufacturer from the delivery of materials from vendors and carriers, to initial receipt of materials, inspection, repackaging, and final placement of these materials in their warehouse locations.
Delphi Packard Electric Systems, General Motors Corporation John A. Sankovich	Taguchi methods were first used to define the impact of extrusion processing factors on fuselink cable processing. These factors were then optimized in order to achieve fuselink with the desired insulation strip force. Then, an OTPM minimum essential information analysis was used to identify all non-value added activities that resulted from poor fuselink quality. These activities were quantified in order to allow management to more accurately measure and eliminate all forms of waste involved in the fuselink production process.

6 Conclusion

Numerous field tests by practicing engineers in a variety of manufacturing companies, as well as consulting engagements conducted by the author have all shown that an OTPM-based IPFA analysis of any complex, repetitive, real-time, object transformation process can identify minimum, necessary and sufficient embedded computer information requirements. Since these requirements are in the form of well-defined input data and output information from and to people, machines, and other embedded computers in a bounded embedded system, their specificity, in addition to process timing requirements, provides a complete specification for developing process

control software. From an industrial engineering perspective, when the control loops are completed and put into operation, the underlying embedded computer software can be easily validated by engineering inspection using the same IPFA methodology.

7 Acknowledgement

The author would like to acknowledge all of his experienced engineering graduate students who have carried out field tests of the OTPM-based methods described in this paper, and also the numerous manufacturing managers and information system software engineers whose helpful comments freely offered in a variety of conferences and workshops helped fine-tune the methodology over the past seven years.

8 About the Author

Dr. Manley is Professor of Industrial Engineering and Director, M.S. Degree Program in Manufacturing Systems Engineering (MSEP), University of Pittsburgh; Member of the Board of Directors Emeritus, Concurrent Technologies Corporation, Chairman, Computing Technology Transition, Inc., and LtCol Regular Air Force, Retired. His prior leadership positions include: Assistant to the Director, The Johns Hopkins University Applied Physics Laboratory; Director, Programming Applied Technology, ITT Corporation; Vice President Engineering and Technology, Nastec Corporation; and Director, Software Engineering Institute, Carnegie Mellon University.

References

1 Manley, John H., "Embedded Systems," in *Encyclopedia of Software Engineering*, John J. Marciniak, Ed., Wiley-Interscience, John Wiley & Sons, Inc., New York, 1994, pp. 454-458.

2 Manley, John H., "Managing Software in the Automated Factory," Chapter in *The Automated Factory Handbook: Technology and Management*, David I Cleland and Bopaya Bidanda, eds., TAB Professional Books, New York, 1990, pp. 198-219.

3 Manley, John H., "OTPM and the New Manufacturing Paradigm," in *Computers and Industrial Engineering*, Vol. 23, Nos. 1-4, Pergamon Press Ltd, Oxford, England, 1992, pp. 427-430.

4 Manley, John H., "Information Process Flow Analysis (IPFA) for Reengineering Manufacturing Systems," in *Computers and Industrial Engineering*, Vol. 25, Nos. 1-4, 1993, pp. 275-278.

5 Ross, D. T, Structured Analysis: A Language for Communicating Ideas, *IEEE Transactions on Software Engineering*, Vol. 3, No. 1, 1977, pp 16-33.

6 Rasmus, Daniel, "Redesigning the Corporation with IDEF's Help, *Manufacturing Systems*, December 1988, pp. 26-31.

7 Leveson, Nancy G., et al, "Requirements Specification for Process-Control Systems," *Information and Computer Science Technical Report 92-106*, University of California - Irvine, November 10, 1992, 54 pp.

8 Rine, David C. and Bharat Bhargava, "Object-Oriented Computing," *Computer*, Vol. 25, No. 10, Oct 1992, pp. 6-10.

9 Ibid, p. 17

10 Manley, John H., *Rise Above the Rest: The Power of Superior Information, Knowledge and Wisdom, Third Edition*, Cathedral Publishing, Pittsburgh, Pennsylvania, 1998.

11 Manley, John H., "Information Process Flow Analysis (IPFA) for Reengineering Manufacturing Systems," Op. Cit.

Defining and Validating Embedded Computer Software Requirements Using the ECS, OTPM and IPFA

23rd Software Engineering Workshop
December 3, 1998

Dr. John H. Manley
University of Pittsburgh
Computing Technology Transition, Inc. (CTTI)
jmanley@engrng.pitt.edu

AGENDA

- Software-Dependent "Real-Time" Systems
- ECS Experience and Research Interest
- Aerospace/Weapon Systems Special Attributes
- Industry "Embedded Software" Acquisition Issues
- Manufacturing Systems Attributes and Tools
- OTPM and IPFA Solution for "Manufacturing"
- Manufacturing Industry Application Results
- Summary and Conclusions

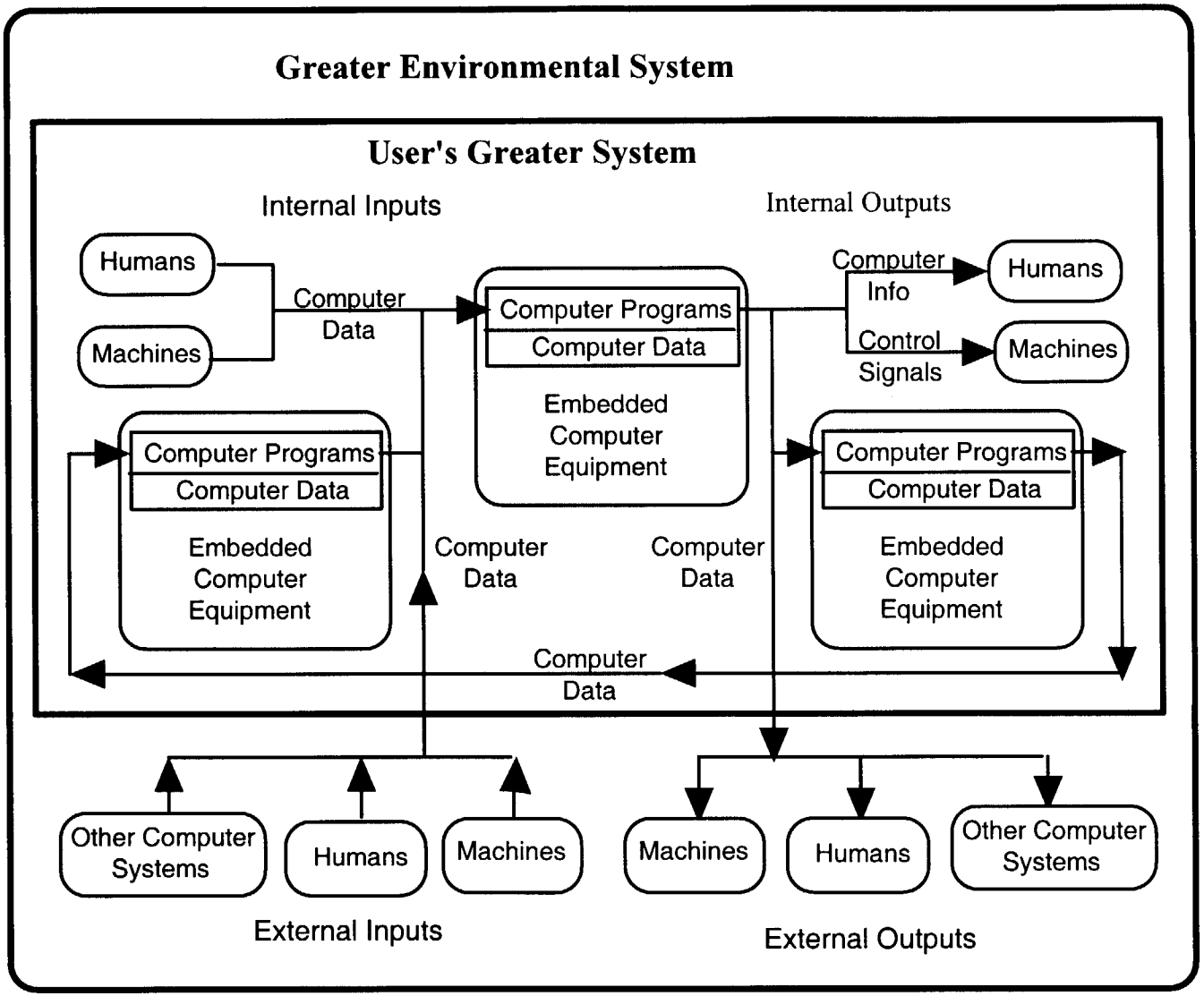
SOFTWARE-DEPENDENT SYSTEMS

- Boeing 777
- U.S. Air Traffic Control System
- Aluminum Sheet Rolling Mill
- Military Command and Control System
- Automobile Assembly Plant
- Space Shuttle Operations Center
- Your New Automobile
- Interactive Videoconferencing System

ECS EXPERIENCE—RESEARCH INTEREST

- **Air Force Career—Lt Col, USAF, Retired**
  - Ammunition/EOD/Nuclear Weapons Loading Officer—SAC
  - Master Navigator, Bomb-Nav Instructor, Flight Examiner
  - Designed and Developed Command and Control Systems
  - Formulated USAF "Embedded Software" Acquisition Policy
- **Civilian Industry and Academic Career**
  - Improved DoD-wide Software Acquisition Policies at APL
  - Directed Software Technology for $2 billion ITT System
  - First Full-time Director, DoD Software Engineering Institute
  - Officer and Director of Software-Related Companies
  - Professor of IE and Manufacturing Systems Engineering

EMBEDDED COMPUTER SYSTEM (ECS) MODEL



SOURCE: Dr. John H. Manley

AEROSPACE WEAPON SYSTEMS ATTRIBUTES

- **True "Real-Time" Operation—Vehicle Keeps Moving**
  - Complex and dynamic mathematical algorithms
  - System architecture must define all human-machine-computer information/communication interfaces
- **DoD Aerospace Systems Require Complex Software for:**
  - Command and control interfaces—open and secure
  - Electronic warfare detection and countermeasures
  - Weapon systems operation and accuracy
  - Adverse weather and terrain avoidance operation

Aircrew Lives Depend on Reliable Software!

MAKING "INVISIBLE" SOFTWARE VISIBLE

- **Known to the DoD as "Invisible Cloth" in the 1970s**
  - Humans and hardware are "visible" physical objects
  - Software involves a series of "invisible" intellectual objects
  - Software development is "invisible" compared to hardware
- **Required Information for ECS Development**
  - "Human" system specifications must include detailed physical profiles and position descriptions
  - "Hardware" system specifications must include detailed architectural and engineering drawings
  - "Computer Program" system specifications must include detailed computer program data input/information output, and information system interface and timing requirements

MANUFACTURING INFORMATION SYSTEMS

High quality information is necessary for (1) effective operation of manufacturing enterprise supporting equipment, facilities and processes, (2) providing customer and supplier services, and (3) proper operation of computers embedded in so called "intelligent" manufactured products. Therefore:

Reengineering any process in isolation, or products as stand-alone entities, is not advisable in today's manufacturing enterprise without also addressing related business, engineering, and other enterprise product and process information systems.

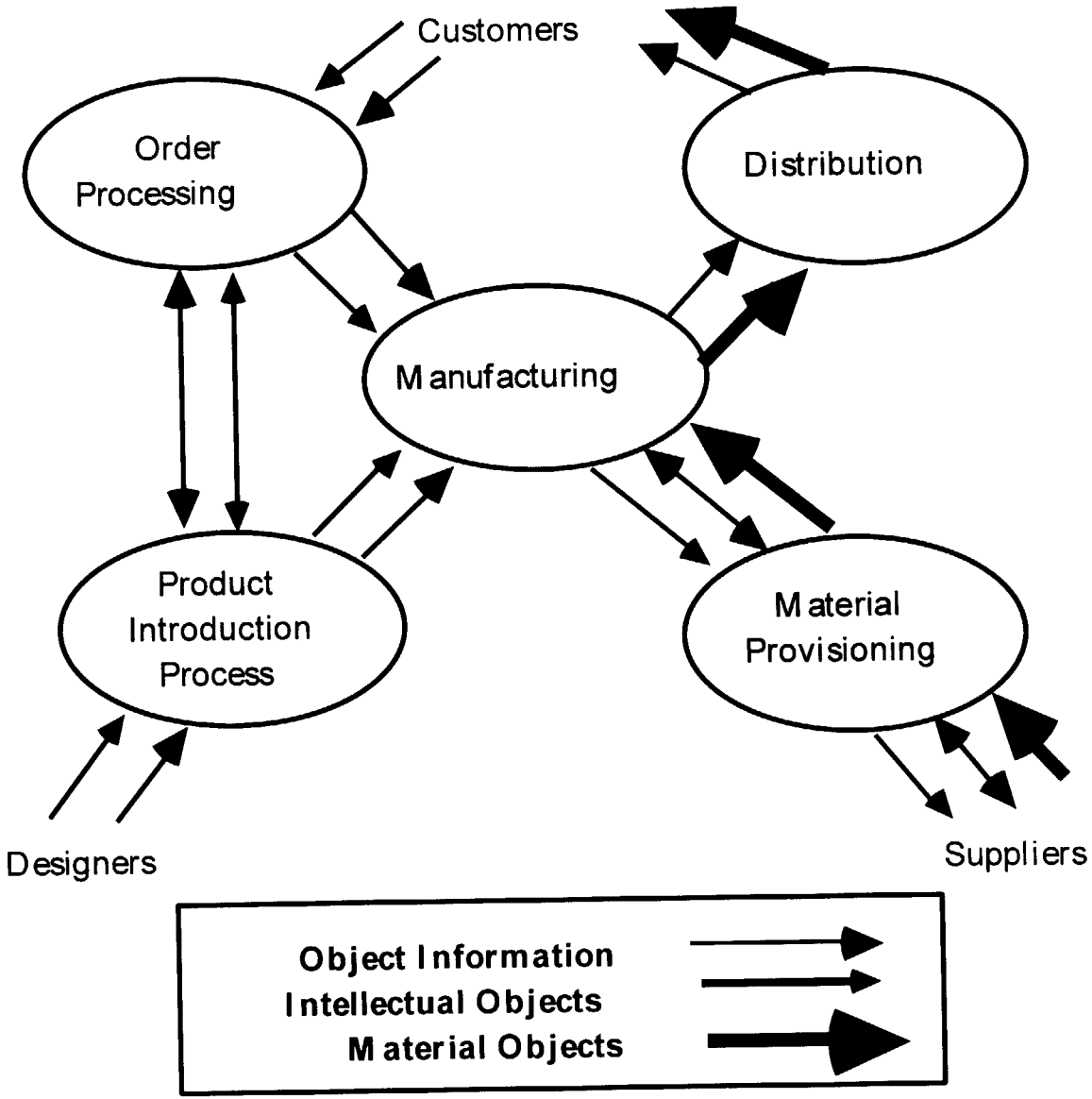
MANUFACTURING SYSTEMS ATTRIBUTES

- **Manufacturing Processes**
  - Repetitive, linked, input-process-output phases
  - Phases transform physical and intellectual objects from one state to another
- **Manufacturing Engineering—Micro Focus**
  - Physical object phase transformations are normally designed by:
    - Mechanical, Electrical, Chemical, and/or Metallurgical Engineers
  - Intellectual object transformations are normally designed by:
    - Industrial, Architectural, and/or Software Engineers
- **Manufacturing <u>Systems</u> Engineering—Macro Focus**
  - Repetitive "earth-to-earth" transformation processes, e.g., automobiles
  - Repetitive enabling processes, e.g., supply chain, finance, management
  - Physical and intellectual object combinations, e.g., automated systems
  - All "engineers" (subject matter experts) are required to work on teams

MANUFACTURING SYSTEMS ENGINEERING DESIGN TOOL EXAMPLES

- **Human component design tools**
  - Job descriptions
  - Ergonomic analyses
- **Machine component design tools**
  - Engineering drawings
  - Simulation models
- **Embedded computer component design tools**
  - Process flow analyses (PFA)
  - Information Process Flow Analyses (IPFA)

TRANSFORMATION PROCESSES AND INFORMATION SYSTEMS



ECS SOFTWARE (RE)ENGINEERING TOOLS

- **Minimum Essential Information (MEI) Guideline:** Remove redundant and/or unnecessary information to simplify control system requirements and insure that necessary and sufficient information is provided where and when required.
- **Embedded Computer System (ECS) Model:** Identify human, computer, and machine data exchange relationships to help define internal and external physical communication requirements.
- **Process Flow Analysis (PFA):** Industrial Engineering method for increasing the efficiency of any real-time repetitive process.
- **Information Process Flow Analysis (IFPA):** During a PFA, identifies extraneous, garbled or missing information necessary for efficient operation of all types of process control loops.
- **Data Design Tools:** Transform high-level IFPA-derived data requirements into programmable data elements.

MINIMUM ESSENTIAL INFORMATION (MEI)

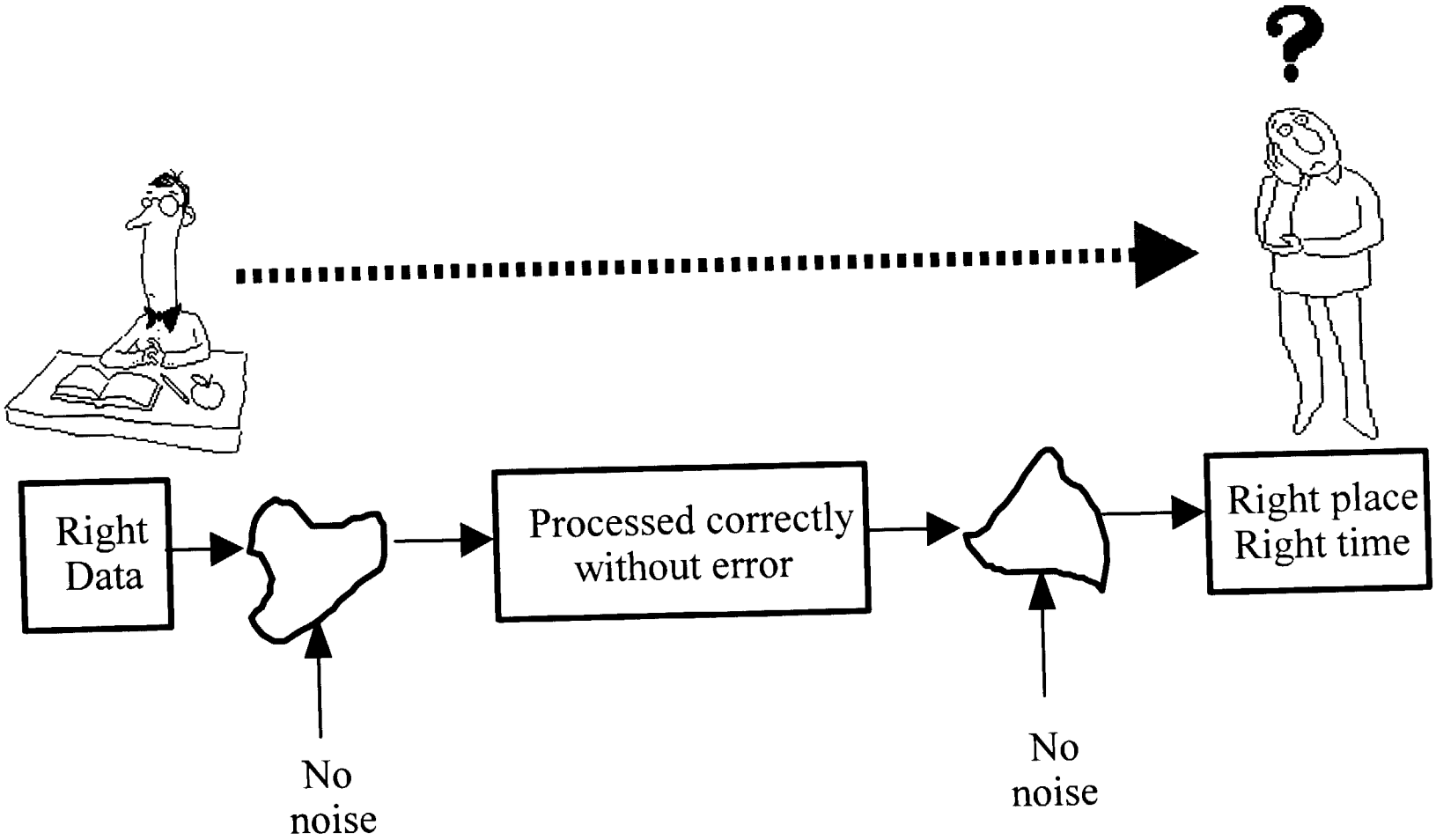
One key to embedded system software engineering success is for *everyone* to actively and continuously identify and reduce the amount of information they generate, use, and store at every point in the life cycle. Recommended guidelines:

- DO NOT retain preliminary process descriptions and similar documentation
- DO NOT hold unnecessary meetings
- DO NOT require or generate FYI reports
- DO eliminate all unimportant information
- DO tell others that you don't need that data
- DO NOT try to manage by paper product alone
- DO develop and use tangible technical progress indicators

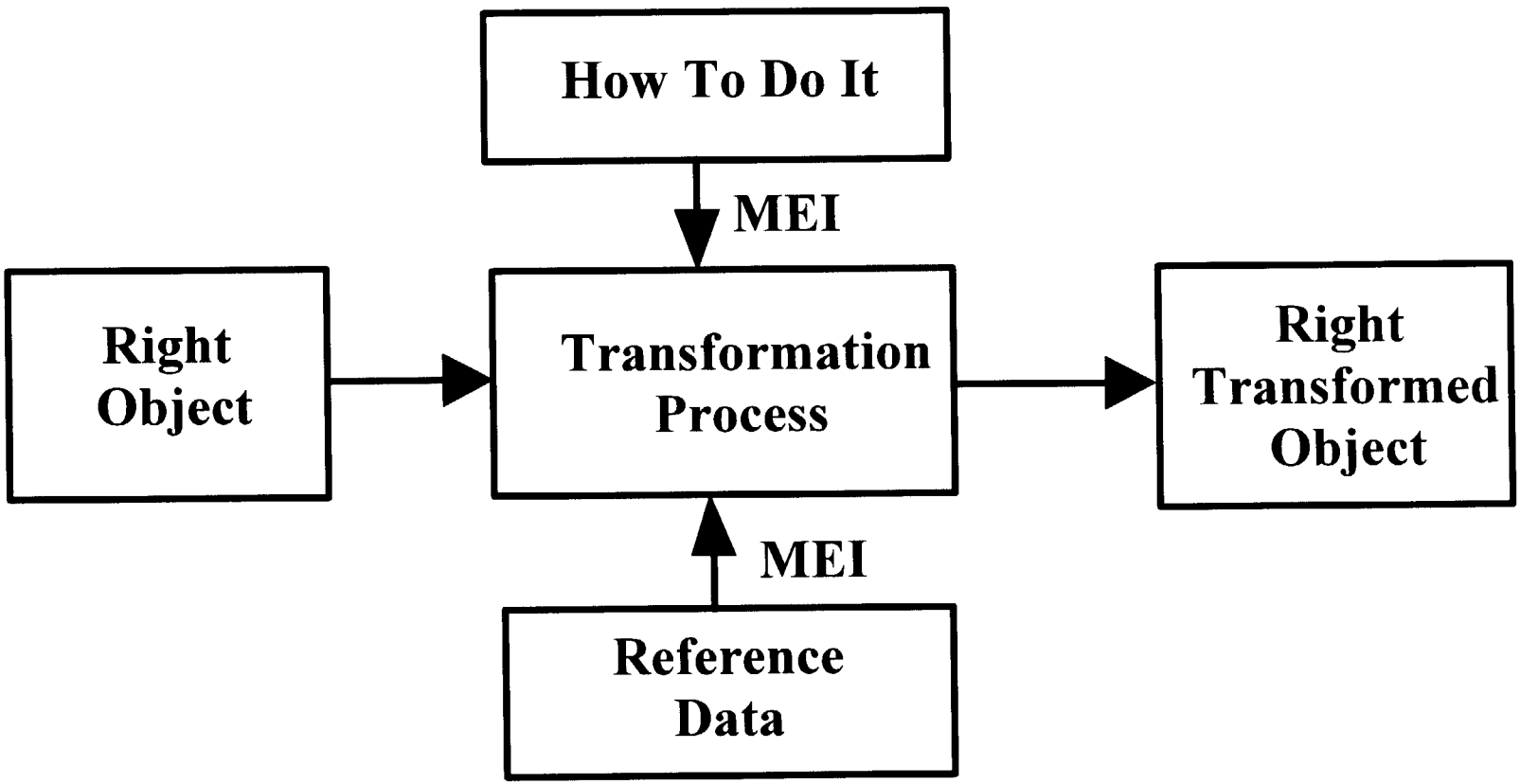
OTPM-BASED ARCHITECTURAL MODELING

- **Object Transformation Process Model (OTPM)**
  - Provides a top-level architectural framework for designing, integrating, troubleshooting, and reengineering enterprise-wide management, finance, engineering, and quality assurance process control information systems.
  - Used as guide for IPFA team members to ensure mutual understanding of reengineering project requirements.
  - Provides comprehensive framework for defining embedded software input/process/output specifications.

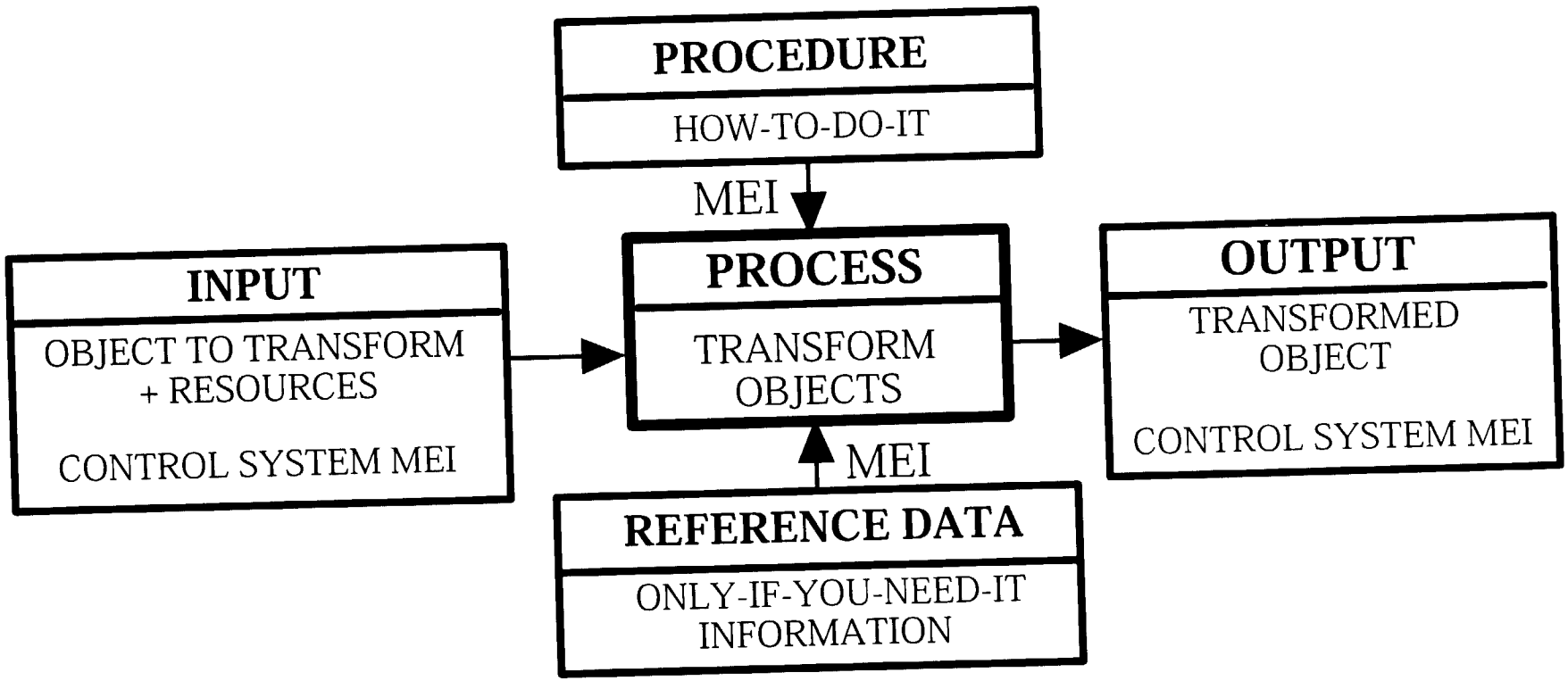
OTPM INFORMATION CONCEPT



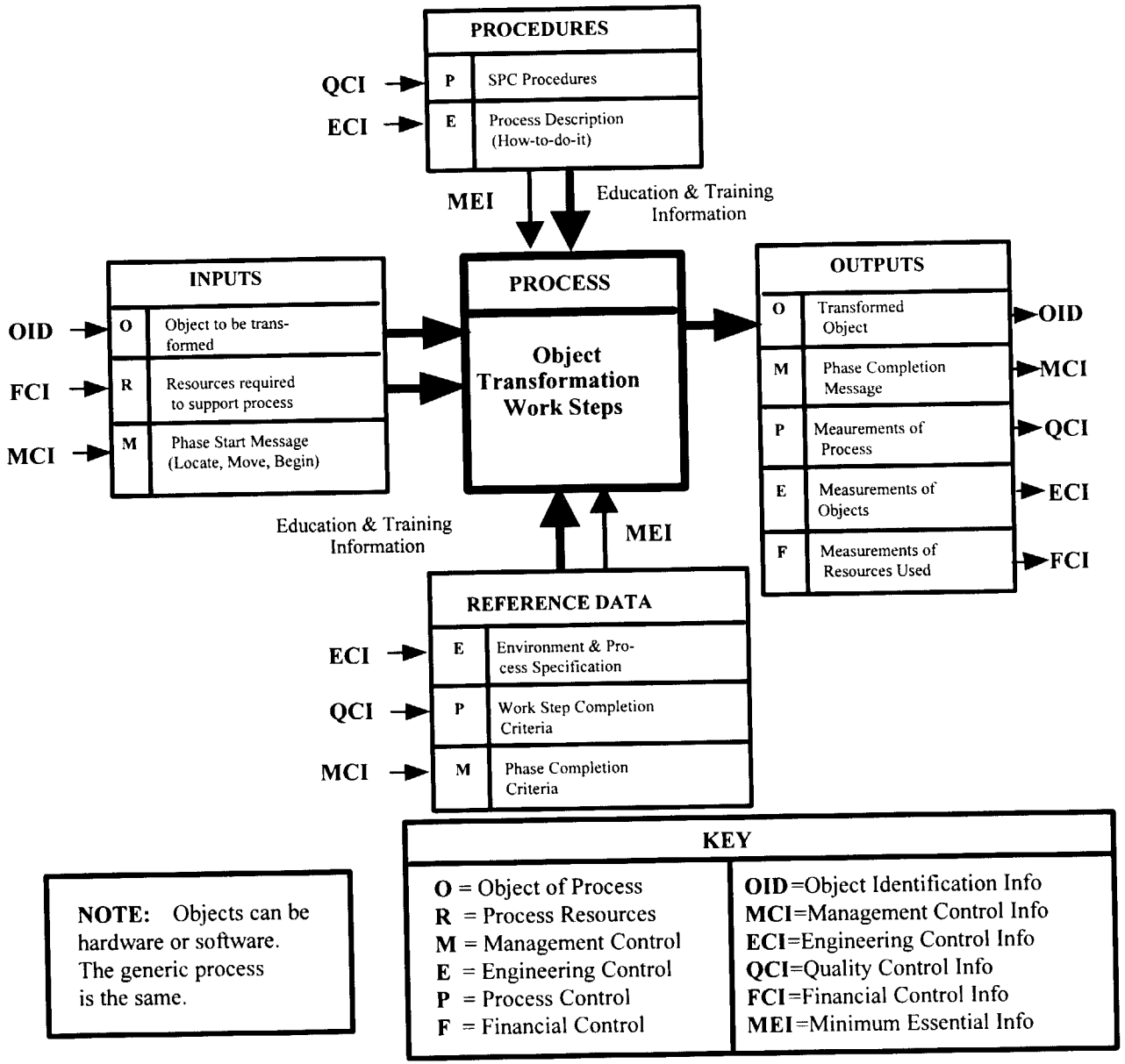
OBJECT TRANSFORMATION PROCESS CONCEPT



OTPM GENERIC MODEL



OBJECT TRANSFORMATION PROCESS MODEL (OTPM)



ECS, OTPM, and IPFA FIELD TEST RESULTS

- **General Motors:** Reengineered plastic molding plant process control information systems—$1.8 million annual saving
- **Bloom Engineering:** Improved reuse of design information for developing metallurgical furnace control systems
- **PDVSA (Venezuela):** Standardized 70 different procedures for developing purchase requisitions for offshore buyers
- **Delphi Packard Electric:** Used OTPM-based information system analysis to solve ignition cable loose core problem
- **H.J. Heinz:** Developed statistical process control system for peach puree manufacturing resulting in zero scrap loss
- **Elliot Company:** Used PFA and IPFA to improve supply chain receiving process for gas turbine generator manufacturing

SUMMARY AND CONCLUSIONS

- Field tests by practicing engineers and consulting by the author in manufacturing companies have shown that an OTPM-based IPFA analysis of any complex, repetitive, real-time, physical or intellectual object transformation process can define MEI and ECS human, machine and embedded computer information requirements.
- The IPFA-derived ECS input/output/procedure/reference information provides system-engineered embedded computer software requirements for reengineering object transformation processes.
- Implementation of IPFA recommendations for software changes by manufacturing companies have all proved to increase their overall efficiency and competitiveness.

(19)

IN-61

Using Automatic Code Generation In the Attitude Control Flight Software Engineering Process

Primary Author: David McComas
NASA Goddard Space Flight Center, Code 582
Greenbelt, MD 20771
301-286-9038
david.mccomas@gsfc.nasa.gov

Co-Authors: James R. O'Donnell, Jr., PhD
Stephen F. Andrews

Abstract

This paper presents an overview of the attitude control subsystem flight software development process, identifies how the process has changed due to automatic code generation, analyzes each software development phase in detail, and concludes with a summary of our lessons learned.

Attitude Control Subsystem (ACS) Flight Software (FSW) and the processes that govern its development are complex. The Microwave Anisotropy Probe (MAP) spacecraft's ACS FSW, currently being developed at the NASA's Goddard Space Flight Center (GSFC), is being partially implemented using Integrated Systems Inc.'s (ISI) MATRIXx which includes an automatic code generation tool AutoCode™. This paper examines the "traditional" ACS FSW development process and describes how the MAP effort, augmented with ISI's tool set, has addressed ACS FSW development complexities.

The MAP ACS team carefully scoped the use of the MATRIXx tools from the outset of the project. The analysts confirmed that MATRIXx was suitable for analysis and algorithm development, but was not certain that AutoCode would be used. Initially, AutoCode's role was to automatically generate an algorithms specification, which has traditionally been a laborious process. If the generated code could be verified, and it passed size and performance requirements, then it would be considered for use as flight code. This low risk approach allowed the team to investigate new technology while addressing the demands of MAP's ambitious schedule with a small development team.

The paper is structured into two sections. The first section provides contextual information for the second section. The first section describes the MAP mission and the flight architecture on which the ACS FSW resides, and the MATRIXx tools. Section two presents an overview of the ACS FSW development process, identifies how the process has changed due to AutoCode, analyzes each software development phase in detail, and concludes with a summary of our lessons learned.

MAP Mission and Flight Architecture Overview

MAP's mission is to probe conditions in the early universe by measuring the properties of the cosmic microwave background radiation over the full sky. These measurements will help determine the values of cosmological parameters associated with the "Big Bang" and determine how and when galactic structures formed. MAP will maintain a halo orbit about the Sun-Earth Lagrange point (L_2), 1.5 million kilometers from the Earth (away from the sun). MAP will maintain a 0.464 rpm spin rate about the spacecraft's Z axis during science observations.

The ACS FSW plays a central role in every phase of the MAP mission using, sensors and actuators to perform attitude determination and control and fault detection and correction. Following launch, the ACS must reduce spacecraft body rates and orient the spacecraft to a power-positive and thermally-safe attitude. The ACS must provide three-axis inertial pointing and provide the capability to slew the spacecraft to new attitudes. The ACS must be capable of doing orbit maneuvers using thrusters. These maneuvers will be used to get to L_2 and to perform L_2 station keeping. The ACS also controls the 0.464 rpm science observation spin.

Figure 1 shows where the ACS FSW fits into the MAP flight architecture. The portion of ACS FSW relevant to this paper resides on the Mongoose processor in the ACS task. The Mongoose uses the software bus for inter-task communication. The software bus is a GSFC custom-built library that provides standardized packet-based inter-task communication and insulates applications from the real-time operating system. A task defines a packet pipe on which to receive data. Task execution is usually controlled by a task pending for data with an optional timeout. The ACS task pends for an Attitude Control Electronics (ACE) sensor data packet, which is generated at a 1Hz rate. After the ACS task receives the packet it converts the sensor data to engineering units in the body frame, updates its attitude knowledge, executes a control law, and issues an actuator command. Each second the ground command packet pipe is polled for new commands, and telemetry packets are generated for onboard storage and/or downlink. Note that the ACS task is isolated from hardware interface details and uses the software bus for all external communications.

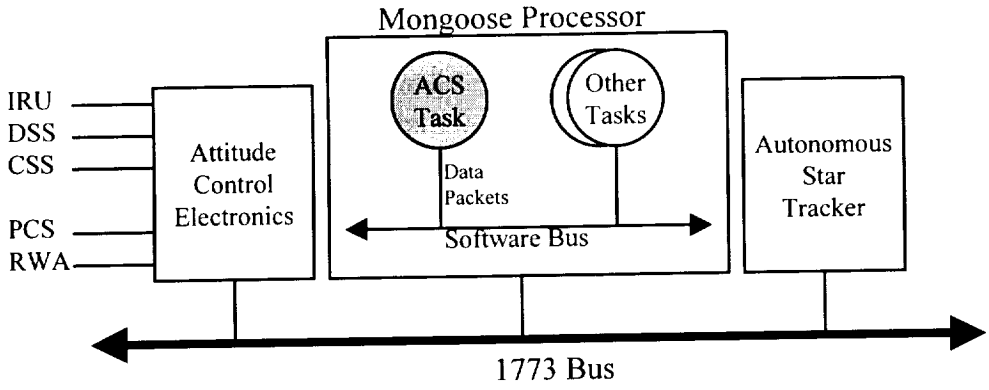


Figure 1 – MAP flight architecture

Two additional ACS features need to be described in order to understand the analysis and design of the automatically generated code and the code that interfaces with it. These two features are the sensors and actuators and the ACS operational modes. MAP uses the following sensors and actuators for attitude determination and control:

Inertial Reference Units (IRU)	Measure changes in MAP's angular position. Spacecraft body rates are derived from the incremental angular measurements.
Digital Sun Sensor (DSS)	Provides accurate measurements ($< 0.01^{\circ}$) of the sun's position within a 64 degree square field of view.
Coarse Sun Sensors (CSS)	Provide coarse measurements ($< 10^{\circ}$) of the sun's position. The CSSs are mounted to provide complete sky coverage.
Autonomous Star Tracker (AST)	Provides an estimated attitude derived from star measurements.
Propulsion Control System (PCS)	Provides external force and torque to the spacecraft via hydrazine-fueled thrusters.
Reaction Wheel Assembly (RWA)	Provides spacecraft attitude control via three reaction wheels.

MAP uses five operational modes to achieve its mission goals. Modes are defined in terms of operational objectives, spacecraft control objectives, and performance criteria. Each mode specifies a set of sensors and actuators and a control subsystem configuration.

MAP defines the following modes:

Operational Mode	Description
Sun Acquisition (SA)	Uses IRUs, CSSs, and the RWA to acquire a sun-pointing, power and thermally-safe attitude within 20 minutes from any initial attitude.
Inertial (IN)	Uses IRUs, DSS, ST, and the RWA to acquire and hold a fixed commanded attitude.
Observing (OB)	Uses IRUs, DSS, ST, and the RWA to perform a scanning pattern. Observing is the only mode used for collecting science data.
Delta-V (DV)	Uses IRUs and the PCS to perform spacecraft maneuvers. Delta-V is used for trajectory management to get to the Sun-Earth L_2 point approximately 1.5 million km from the Earth (away from the sun) and for L_2 station-keeping.
Delta-H (DH)	Uses IRUs and the PCS to perform momentum unloading.

MATRIXx Overview

The purpose of this paper is not to evaluate MATRIXx. However, it is necessary to understand some of the components and features of MATRIXx in order to understand how they impacted the software development. Figure 2 shows the MATRIXx runtime environment. SystemBuild provides a graphical environment for building models and a graphics package for analyzing data. Xmath provides text-based windows for user interaction and the mathematical computation engine. The three standard user windows are status log, command input, and error message.

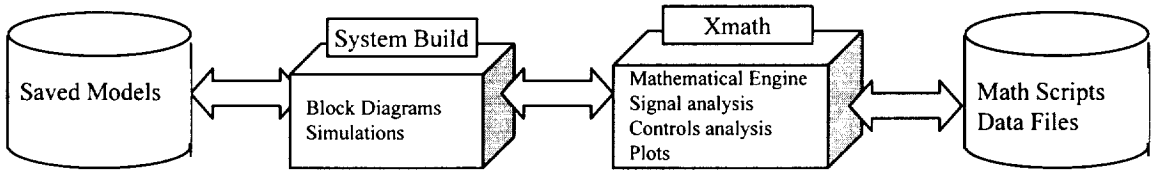


Figure 2 – MATRIXx Environment

An engineer graphically decomposes complex models in SystemBuild using SuperBlocks. SuperBlocks are characterized by their inputs, outputs, and user defined attributes. Timing attributes include continuous, discrete, procedure, and triggered. Procedure blocks can be further classified as standard, inline, macro, interrupt, background, or startup. Attribute details will be described as needed in the paper.

SuperBlocks may contain other SuperBlocks and/or functional blocks. Hierarchies of SuperBlocks are used to abstract system details. Functional blocks cannot be further decomposed. The most common functional blocks used on MAP include trigonometric functions, algebraic functions, logical functions, and dynamic systems functional blocks. Most functional blocks are "closed" which means an engineer can define the block's I/O and configuration parameters, but cannot change the block's functionality. For MAP, we used three types of "open" blocks, which allow the user to extend the system's functionality. Algebraic blocks allow the user to define the block's outputs as algebraic functions of the block's inputs (and other parameters). BlockScript blocks allow the user to use BlockScript, a FORTRAN-like procedural language, that allows many programming constructs. Finally, User Code Blocks (UCB) allow user supplied C code to be linked with SystemBuild.

In addition to providing functional organization, SuperBlocks also control some aspects of the data flow. There are three basic methods for transferring data with a SuperBlock. SuperBlock I/O pins can be used to pass data up and down a SuperBlock hierarchy. Read-from and write-to variable blocks can input and output data from a global workspace. The last method, available for most SuperBlocks, is to define SuperBlock parameters. Parameters are variables that are imported from the Xmath variable space and are called percent variables (%VARs). %VARs are used to define variables, such as alignment matrices and controller gains, that do not change during a simulation. AutoCode converts %VARs to global variables that can be written to and read by code external to the automatically generated code. %VARs can be logically grouped into partitions and a SuperBlock can specify a particular partition for its %VARs.

The automatic code generation process is shown in Figure 3. A model must be loaded into SystemBuild and default values for %VARs should be established. This is best achieved by using a MathScript (Xmath command files) to define the defaults. The top most SuperBlock is supplied to AutoCode and code is generated for this SuperBlock and every subordinate SuperBlock. Code can be generated for multi-rate systems using the scheduled-subsystem option, and for single rate systems using the procedures-only option.

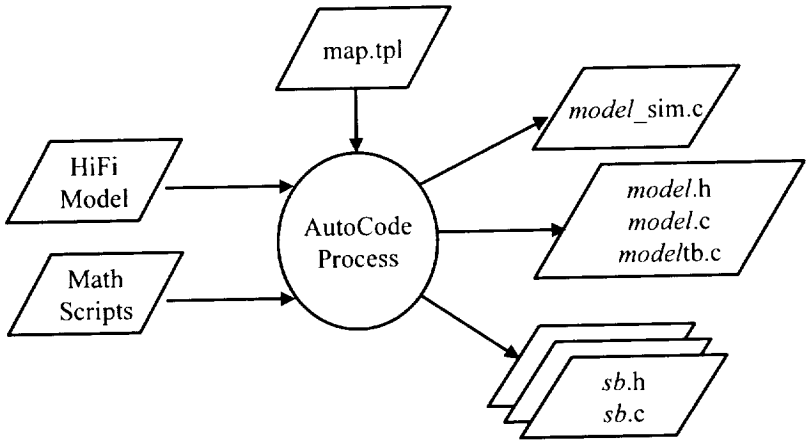


Figure 3 – Automatic code generation process

The code generation process is controlled by a script written in ISI's Template Programming Language (TPL). We extended ISI's default TPL file (renamed to map.tpl) to include the following features:

- Non-FSW is output to a separate *model*_sim.c source file. This includes code such as UCB wrappers (code used to interface C language functions into a simulation via a User Code Block) that could be used in a simulation environment but are not needed in the flight environment.
- Separate header and source files are created for each non-inline SuperBlock. This enhances readability, encapsulation, maintenance, and configuration management. Late changes will result in individual files being delivered.
- Two interface functions *model*_Init() and *model*_Dispatch() are created to provide a consistent interface to the automatically generated code. These functions also provide a placeholder for customizations.

The MATRIXx AutoCode process is essentially closed, which means the code generated for a functional block cannot be altered. This is why the BlockScript and Algebraic blocks are considered to be open since they allow user customization of the generated code. As an example of the closed nature of the code generation process, let us consider how the %VARs are defined. To define the %VARs the TPL script calls a predefined TPL function named define_percentvars(). This generates the code for all of the %VAR definitions. The TPL script can add code around the %VAR definitions and it can control what file the definitions appear in, but that is all it can do.

ACS FSW Development Process

Figure 4 illustrates the MAP ACS FSW development process. The solid lines represent the traditional process and the dashed lines indicate where the process has significantly changed for MAP. Many traditional parts of the process were impacted as well, and these

impacts will be described in their respective sections. Note there is an iterative aspect to our development process that is not shown in the figure. The FSW is delivered to the build test team in incremental builds and the requirements and design are continually refined throughout the process.

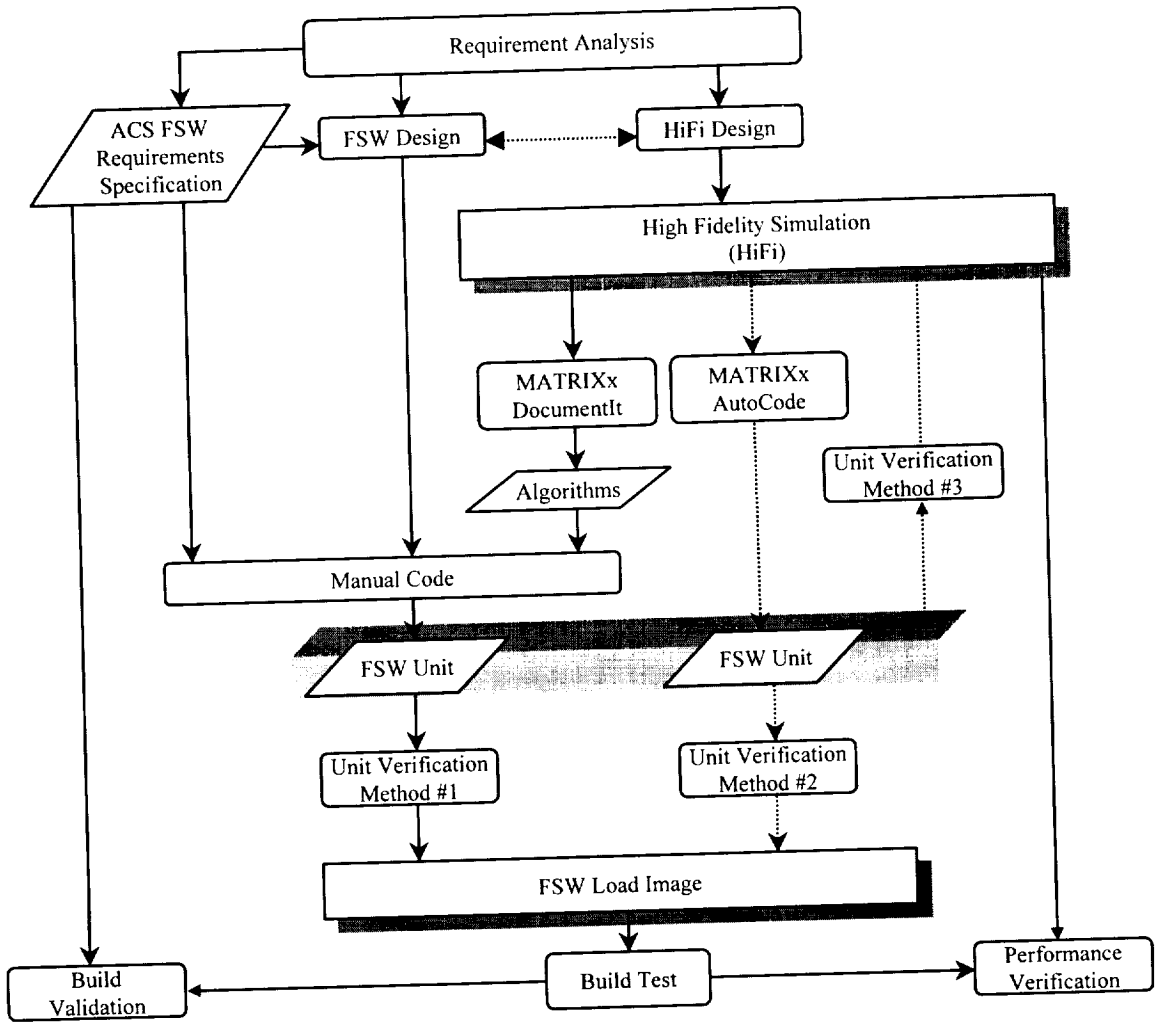


Figure 4 - ACS FSW Development Process

The process begins with ACS FSW requirements analysis. This is a system engineering process primarily involving Guidance Navigation and Control (GNCC) analysts, FSW specialists, and spacecraft engineers. This activity produces an ACS FSW Requirements Specification and feeds directly into both the FSW and high-fidelity (HiFi) simulator designs. HiFi is required for the GNCC analysts to validate the spacecraft controls algorithms which are needed in the ACS FSW. On previous GSFC missions, the ACS FSW architecture and the HiFi software architectures were developed independently, with minimal-to-none commonality between the two software systems.

The use of AutoCode requires that the FSW and HiFi architectures account for one another's environment. The graphical HiFi environment encapsulates function and data into a component called a SuperBlock. In order for a translated SuperBlock to exist in the FSW environment, HiFi must emulate the FSW environment or the SuperBlock must avoid interfacing or relying on features of the FSW environment. Any differences that exist between the two environments must be accounted for by the automatic code generation process or by manually changing code after it has been generated. The analysis and design section describes the translation strategies that were taken on MAP.

The implementation phase gets its inputs from the ACS requirement specification and the ACS algorithm specification. The ACS requirements specification defines what the FSW needs to accomplish in terms of functional and performance statements. The algorithms specification is a companion to the requirements and it defines the mathematical details needed to be implemented by the FSW in order to meet the functional and performance criteria. Traditionally all of the FSW has been manually coded from these inputs. Two significant changes were made on MAP. First, the generation of the ACS algorithms specification was automated using ISI's DocumentIt™. In the past, the generation and maintenance of the ACS algorithm specification was a tedious job that has required a dedicated analyst. Second, the automatic generation of some of the flight code eliminated part of the manual coding effort. These two changes were not free and the price of using the tools will be discussed in the implementation section.

Testing occurs at both the unit and build test levels. At the unit level, 3 verification processes have been enumerated. Unit verification method #1 requires the developer to verify that the requirements and algorithms have been implemented correctly. Depending upon the scope of the algorithms being coded, unit test data from HiFi may or may not be supplied. Unit verification method #2 is new and is used to verify the automatic code. Since the HiFi and FSW have a common interface to the automatic code it was relatively easy to capture HiFi data at the common interface and feed it through the automatic code. Even without AutoCode, this method could have been employed in the past if the FSW and HiFi designs used a common architecture. Unit verification method #3 was an unanticipated benefit of using the tool set. We were able to take the entire FSW attitude determination subsystem flight code and run it in the HiFi.

At the build test level, the changes were mostly due to other features of the tool set beyond AutoCode. The main advantage of the tool set is that the analysts used the HiFi data analysis platform for test data comparison between data from the development lab and data from the HiFi. In the past, the build test data analysis platform was not necessarily the same as the HiFi's platform. Having the data analysis platform integrated with the HiFi platform also enabled the script files developed during analysis to be used for build test verification.

Analysis and Design

The analysts and programmers had to coordinate their efforts during the analysis phase because automatic code generation requires a tight coupling between the HiFi design and the ACS FSW design. First, both groups needed to understand the capabilities of the tools in order to understand how to best utilize them. Concurrently, a preliminary architecture that would be common to both the HiFi and the FSW needed to be defined. The definition of this architecture was based on MAP requirements analysis and on previous mission architectures. The common architecture identified major components such as sensors, actuators, control modes, attitude determination, and ephemeris and the data that flows between the components. With detailed knowledge of the tools and a preliminary architecture, the MAP team was prepared to define the scope of the automatically generated code. Business forces as well as technical forces shaped the boundary of the automatically generated code.

The primary business issues are that we meet the schedule while delivering a high quality, testable product that implements the requirements. MAP is Goddard's first mission to use an automatic code generator for its FSW, and prior to MAP, Goddard had no experience with ISI's code generator. Our strategy towards mitigating risks was to minimize the impact of the FSW environment upon the HiFi and to limit the scope of the automatically generated code to a portion of the ACS FSW that has a high algorithm-to-code ratio. This strategy minimizes the impact to the analysts while taking advantage of the biggest benefit of AutoCode, which is to eliminate the error prone process of manually translating algorithms to flight code.

Technically, automatic code generation is the translation of a design from the HiFi environment to the FSW environment. The HiFi environment must model any aspects of the FSW environment if the generated code is to be linked with the flight code without any manual changes to the automatic code. As described in the MAP flight architecture overview, MAP is using the software bus as the inter-task communication medium. Opting not to model the software bus in HiFi, we immediately limited the scope of each AutoCode invocation to an intra-task scope and ISI's real-time operating system, pSOSystem, was not even considered. There are also unique flight software interfaces for processing ground commands, generating asynchronous event messages, and notifying the Fault Detection and Correction (FDC) subsystem. Again, we opted not to model these interfaces in the HiFi, further restricting the AutoCode scope.

With these guidelines in hand, we were prepared to identify the portion of the ACS FSW that is automatically generated. Figure 5 shows a simplified high-level block diagram of MAP's flight control software. This software is suitable for a single task because all of its components execute at 1Hz and have fairly strong data cohesion. Sensors measure spacecraft position and rates. Attitude determination uses sensor measurements to update the onboard estimated attitude, which is supplied to the controller subsystem. The desired spacecraft attitude is either supplied by mode management or internally computed by command generation. Attitude error computes control errors for the control law based on a combination of sensor measurements, estimated attitude, and commanded attitude. The control law computes control torques which are output to the actuators.

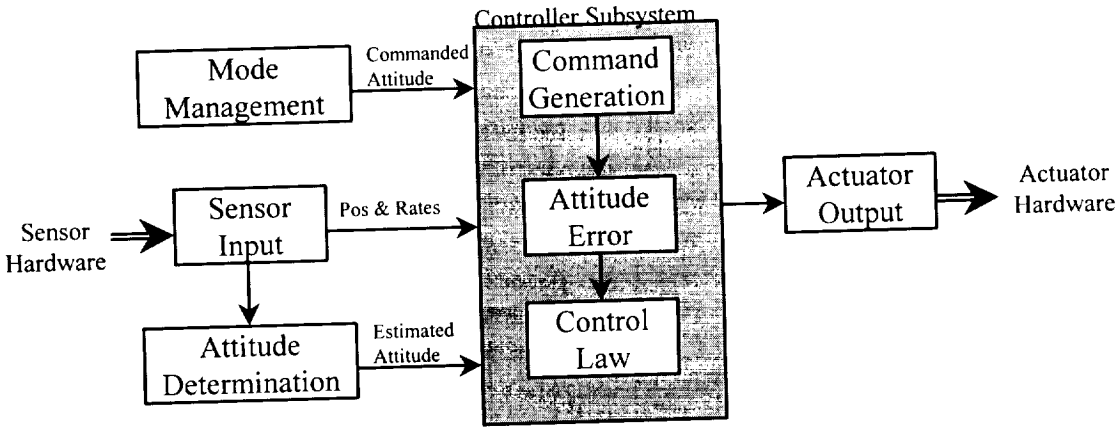


Figure 5 – ACS FSW block diagram

The shaded controller subsystem identifies the portion of the MAP ACS FSW that is being automatically generated. This final design takes advantage of the controller subsystem's relatively small and simple set of inputs and outputs. The controller subsystem's local rotating-sun-reference coordinate frame is encapsulated entirely within the AutoCoded portion of the FSW. Since the controller subsystem components execute at the same rate, we can use the "procedures-only" AutoCode option, which greatly simplifies the automatic code. Attitude determination shares many of the same attributes as the controller subsystem with respect to being suitable for AutoCode, but it was not chosen to be automatically generated since MAP could adapt an existing attitude determination subsystem from a previous mission.

Implementation

Once the scope of the automatically generated code was defined, we turned our attention to the code that is generated and to the manual code that interfaces with the automatic code. Figure 6 is a class diagram showing the classes involved with the manual-to-automatic code interface. AutoCode creates a collection of C functions that can be conceptualized as a single object. This object, named achifi, corresponds to the parent SuperBlock supplied as an input to AutoCode. Achifi provides two interface functions achifi_Initialize() and achifi_Execute(). Outputs from achifi are exported via a global data structure named achifi_Output. The manual code treats this as a read-only data structure; although no mechanisms enforce this rule.

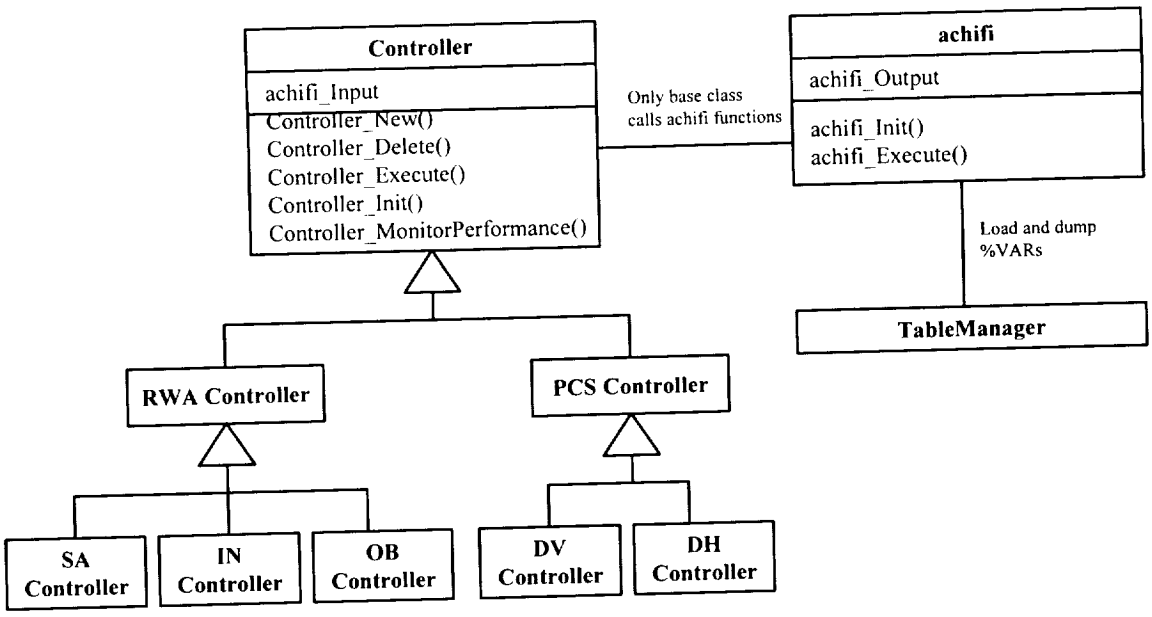


Figure 6 – Manual and automatic code class diagram

The manual code is designed as a class hierarchy designed according to the operational modes identified in the mission overview section. The base controller class defines functions and data that are common to all controllers. Next, the control modes are classified according to which actuator is used for control. Finally, individual modes are defined. The base controller class is the only class that invokes achifi's member functions. Achifi_Initialize() is called when the processor is initialized and whenever a mode transition is performed. Achifi_Execute() is called during each control cycle. Achifi_Execute() manages calling the other functions created by AutoCode.

The object-oriented design was implemented in C by constructing virtual function pointer tables and the design has proven to be very robust with regard to automatic code interface modifications. Most automatic code dependencies are encapsulated by the base controller class, so changes to the interface have had no ripple affect. The base class also provides functionality that can be shared by all controllers. Controller_MonitorPerformance() monitors the performance of achifi by monitoring body rates, attitude errors, and body rate errors. Controller unique performance limits are supplied to Controller_New() when a controller is instantiated. Another benefit of the object-oriented design is that the class hierarchy has resulted in small, easy to test functions. Below are some excerpts from the ACS FSW showing how the base controller class manages the automatic code.

```
Controller file scope excerpt
#include "achifi.h"                    /* Autocode header file  */
achifi_Input_Rec  achifi_Input;        /* Autocode input record */
```

```
Controller_Init() excerpt
achifi_Init(&achifi_Input,TRUE);    /* TRUE means copy %VARs from EEPROM */


Controller_Execute() excerpt
achifi_Input.BodyEstRateX = DataMgr.Config.BodyRate.Comp[X];
achifi_Input.BodyEstRateY = DataMgr.Config.BodyRate.Comp[Y];
achifi_Input.BodyEstRateZ = DataMgr.Config.BodyRate.Comp[Z];

achifi_Input.Rwa1MeasTachSpeed = RWA_ProcData.Speeds[0];
achifi_Input.Rwa2MeasTachSpeed = RWA_ProcData.Speeds[1];
achifi_Input.Rwa3MeasTachSpeed = RWA_ProcData.Speeds[2];
. . .
achifi_Execute(&achifi_Input,DataMgr.Config.DeltaTime);

AttCtl.BodySysMom.Comp[X] = (float)achifi_Output.BodyMeasSystemMomX;
AttCtl.BodySysMom.Comp[Y] = (float)achifi_Output.BodyMeasSystemMomY;
AttCtl.BodySysMom.Comp[Z] = (float)achifi_Output.BodyMeasSystemMomZ;
AttCtl.BodySysMomMag = (float)Vector3f_Mag(&AttCtl.BodySysMom);
. . .
```

The automatic code is less readable than the manual code shown above, but this is mostly due to the inclusion of SystemBuild's numeric block identifiers in the variable names. The automatic code generation is very systematic and once you get a feel for how status information, state information, inputs, outputs, and initialization are managed, the code is is relatively easy to read. In fact, comments are inserted in the code to identify which block is being coded. The code's readability is further enhanced by having each SuperBlock output to a separate file, using data naming conventions in the HiFi, and labeling all block I/O lines in the HiFi.

There are a few drawbacks to the automatic code, but none of them have proven to be fatal. The automatic code is large and less efficient than its manual equivalent. We have not had the luxury of duplicating the coding effort manually, but there have been a couple of cases when a fair comparison between the manual and automatic code could be made. In these cases the automatic code has been two to three times larger than the manual code. This code bloat is primarily due to the fact that AutoCode does a lot of data copying before and after calling a procedure. Declaring procedure blocks inline can reduce this overhead, but this doesn't allow functions to reside in individual files. Since MAP has a 1Hz ACS task execution rate and 4 megabytes of EEPROM, resources have not been a concern.

There are a few non-resource issues that did create some trouble. We wanted to treat %VARs as one or more FSW tables. A FSW table must contain physically contiguous data and we typically achieve this by defining a table as a C structure. Unfortunately the Xmath variable partitions do not translate into C structures. We were able to contiguously group the %VARs by defining the %VARs in a separate file. Using linker scripts, the %VAR object file was defined as a contiguous data structure.

There have been two cases when the generated code wasn't what we expected. In both cases, default values were hard coded for %VARs instead of variables being used. ISI's

response was that these are features of the blocks in question, although that documentation has not been found yet. Hard coded %VARs is unacceptable so we had to develop workarounds. In one case we coded the block in BlockScript and in the other case we changed the design so the offending block wasn't used. Aside from the %VAR problem, the generated code has properly implemented the HiFi design.

The last issue with the generated code concerns the time between valid control cycles, which we refer to as delta time. Nominally, delta time is one second, but there are situations in which this time can be greater than one second. The ACS FSW must use the actual delta time to safely control the spacecraft. However, AutoCode hard codes a one second delta time because the top-most SuperBlock is defined as a 1Hz procedure block. To correct the situation the automatically generated code must be manually changed to use the delta time passed to achifi_Execute(). Since our automatic code generation process produces a separate source file for each SuperBlock, a manual code change has to be made only once, unless the SuperBlock changes in a future build.

Testing

Both unit testing and build testing have been improved by using MATRIXx. Many of these gains are the result of the entire tool set, not just AutoCode, and are also the result of the FSW developers and analysts working more closely together than on previous Goddard missions.

Figure 7 shows the primary unit testing effort. The HiFi outputs simulation results, simulation inputs, and the automatic code to the unit test platform. The unit test (UT) driver is linked with the FSW controller classes and the automatic code. Five HiFi test cases, one for each operational mode, were used as the test suite. This data was run through the FSW and the FSW results were compared to the HiFi results. This testing has consistently shown that the automatic code accurately represents the Systembuild design.

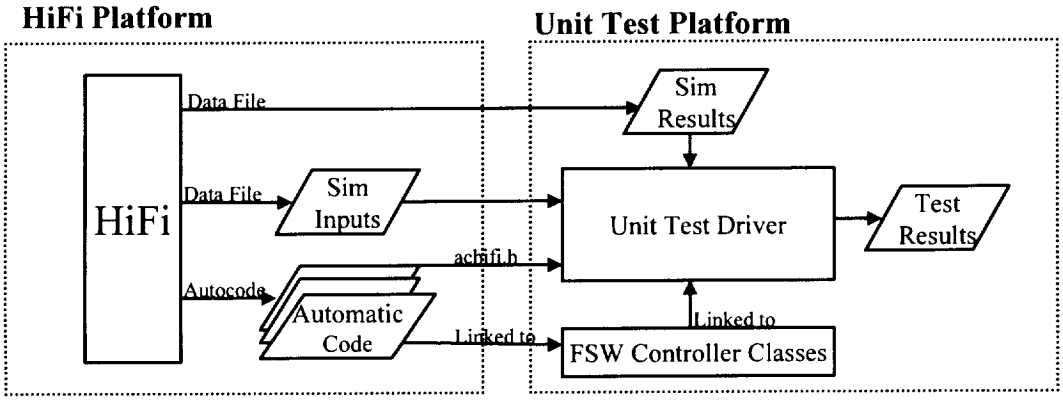


Figure 7 – Automatic code unit test

Figure 7 identifies the primary unit testing activity, but there was additional unit testing performed. A separate UT was developed to perform boundary testing, full path coverage, and to test miscellaneous items such as whether %VARs are properly parameterized. This additional testing is important because the analysts have not traditionally programmed for flight and have not had to worry about issues such as protecting against dividing by zero. This is also our first experience with AutoCode and we don't have past experiences to provide a basis for confidence in the tool. This is the testing that found that the %VARs are not being parameterized for some of the blocks.

Several additional features of the UT process should be noted:

- The UT calls the controller object functions in the same manner as the FSW component that manages the controller objects. This testing verifies that the manual code and the automatic code are properly integrated.
- Achifi.h is shown as a separate input into the UT because it is parsed by the UT to identify the variable offsets within the simulation data file. This aspect is useful because this allows the order of the data within the simulation data file to change without impacting the UT. Since the simulation data file contains data used by the analysts to verify the HiFi, this feature allows the analysts to change what data gets captured without worrying about preserving the order of the data for the UT.
- Since MathScripts managed and documented HiFi test cases, the FSW developers didn't require as much of the analyst's time as in the past to get necessary information. Defining common controller interfaces in both the HiFi and the FSW and having a standardized data file format were the drivers that empowered the UT. These activities could occur with or without automatic code generation.

The improvements made to build testing were achieved by extending the tool set. Build testing involves verifying the FSW's performance in the breadboard lab. Traditionally, a small set of performance test cases were identified. These test cases were executed in both the HiFi and in the build test lab. Two common problems with performance verification involves setting up the same test case in both environments and comparing the results. Through a combination of UNIX scripts and MathScripts, an automated process was developed to transform build test scripts and data sets into HiFi MathScripts. The MathScripts are executed to generate HiFi performance data for the corresponding build tests. The ability to be able to generate these tools was also facilitated by the fact that the HiFi design is similar to the FSW design so the HiFi can be configured in a manner similar to how the FSW is configured via table loads and commands. Another set of MathScript tools were created to automate the generation of comparison plots. The plot generation and comparison tools use GUI-based menu systems making them easy to use. These tools provide a consistent and efficient means for all team members to generate and analyze data without having to dedicate an engineer to these tasks.

Lessons Learned

The most dramatic change has occurred to our ACS FSW development process. Figure 8 shows the same software development process shown in Figure 4 but in terms of activities and products. What has changed is that many people are performing multiple roles. The analysts have participated in every activity. They perform time domain analysis using the HiFi, help to write the requirements, attend code reviews held by the developers, write build test procedures, and serve as the focal point for performance verification. Likewise, the developers have help to write the requirements, reviewed the portion of the HiFi that is being translated to FSW, developed code, and supported performance verification. We have intentionally not allowed the developers to participate in build testing for independent verification. The tools have brought the team closer together but also allowed them to work more independently and efficiently. Many process participants have worked as system engineers with a specialty.

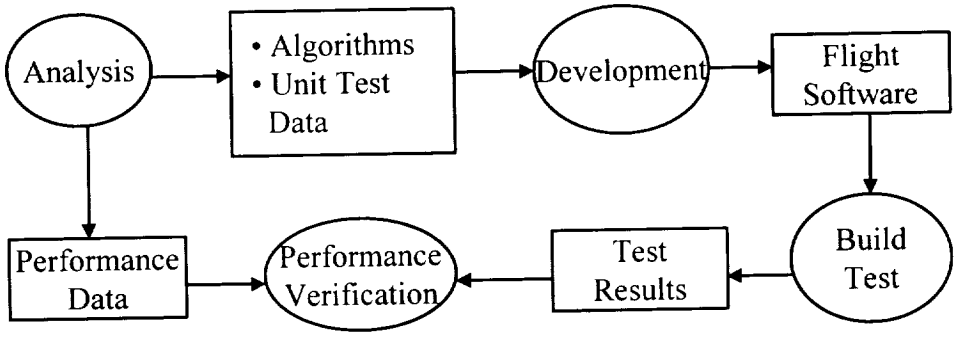


Figure 8 – Software development roles

Trying to quantify these perceived benefits is not so clear cut. We only have limited metrics for the Rossi X-Ray Timing Experiment (RXTE), spacecraft which is a recent Goddard ACS FSW development effort. The following data is a simple comparison of the RXTE and MAP ACS FSW development efforts.

Lines Of Code(LOC)	Spacecraft	Man Years (MY)	LOC/MY
33,318	XTE	13.8	2414
17,525	MAP	6.1	2872

This data appears to show that MAP has been slightly more productive. However, there are many factors to consider when evaluating this data.

- We only have metrics on how much time a developer spent on a project. These metrics are not refined enough to know how much time a developer spent on activities other than development.

- Conversely, the man year data doesn't include analysts or build testers, yet one of the empirical benefits of AutoCode is that it improves the entire development effort. We definitely reduced the manpower required to translate the HiFi to FSW by using AutoCode, and on RXTE, an analyst was dedicated to creating, executing, and plotting HiFi performance runs for build test verification. These benefits have not been quantified by the data above.
- As stated before, the automatic code was two to three times larger than if it were manually coded. The 5,356 lines of automatic code (31% of the MAP ACS FSW) inflate the MAP production rate.

This data is very encouraging considering that MAP incurred a learning curve with the new tool set. Unfortunately we don't have any build test metrics, but the test effort has been observably better than on RXTE and we do know the lab is vacant on weekends!

In summary, our low risk approach was successful in allowing us to investigate new technology while meeting the demands of MAP's ambitious schedule with a small development team.

Using Automatic Code Generation in the Attitude Control Flight Software Engineering Process

David McComas
Stephen Andrews
James O'Donnell,Jr., PhD

12/3/98

Agenda

- Background
  - Microwave Anisotropy Probe (MAP)
  - Attitude Control Subsystem (ACS)
  - MATRIXx tool set
- Software Development Process
  - Analysis and Design
  - Implementation
  - Testing
- Lessons Learned

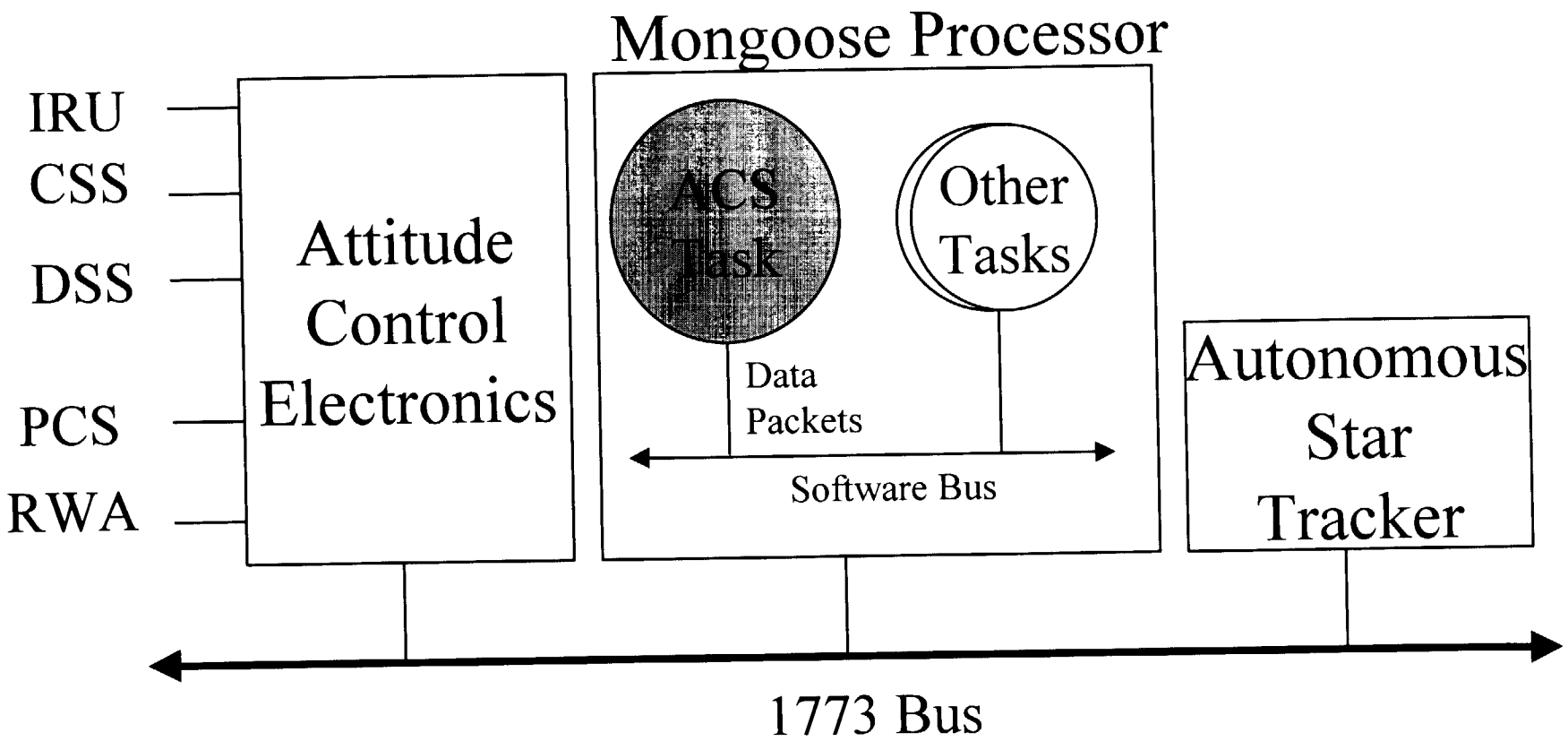
What is the Microwave Anisotropy Probe (MAP) ?

- Spacecraft to measure properties of the cosmic background radiation over the full sky
- Measurements will determine
  - "Big Bang" parameters
  - How and when galactic structures formed
- Maintain a halo orbit about the Sun-Earth Lagrange point (L_2) 1.5 million km from Earth
- Maintain a 0.464 rpm spin rate for science data collection

What is the MAP Attitude Control Subsystem (ACS) ?

- Onboard hardware and software responsible for
  - Attitude Determination
  - Attitude Control
  - Failure Detection and Correction
- MAP ACS manages
  - Control following separation from the launch vehicle
  - Orbit maneuvers to get to L_2 and to maintain L_2 orbit
  - Control of the 0.464 rpm scan
  - Momentum unloading

MAP ACS Flight Architecture



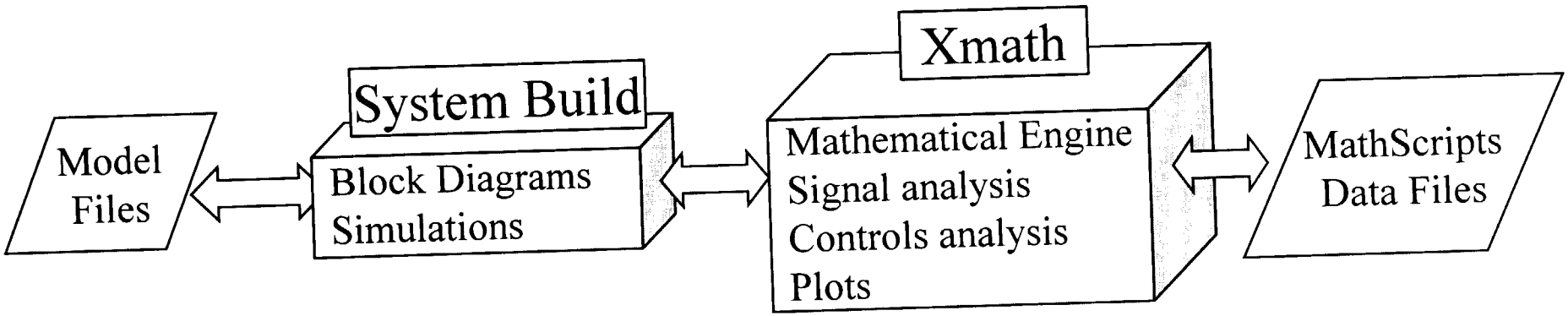
MAP ACS Sensor and Actuators

Inertial Reference Units (IRU)	Measure changes in MAP's angular position. Spacecraft body rates are derived from the incremental angular measurements.
Digital Sun Sensor (DSS)	Provides accurate measurements ($< 0.01^{o}$) of the sun's position within a 64 degree square field of view.
Coarse Sun Sensors (CSS)	Provide coarse measurements ($< 10^{o}$) of the sun's position. The CSSs are mounted to provide complete sky coverage.
Autonomous Star Tracker (AST)	Provides an estimated attitude derived from star measurements.
Propulsion Control System (PCS)	Provides external force and torque to the spacecraft via hydrazine-fueled thrusters.
Reaction Wheel Assembly (RWA)	Provides spacecraft attitude control via three reaction wheels.

MAP ACS Operational Modes

Operational Mode	Description
Sun Acquisition (SA)	Uses IRUs, CSSs, and the RWA to acquire a sun-pointing, power and thermally-safe attitude within 20 minutes from any initial attitude.
Inertial (IN)	Uses IRUs, DSS, ST, and the RWA to acquire and hold a fixed commanded attitude.
Observing (OB)	Uses IRUs, DSS, ST, and the RWA to perform a scanning pattern. Observing is the only mode used for collecting science data.
Delta-V (DV)	Uses IRUs and the PCS to perform spacecraft maneuvers. Delta-V is used for trajectory management to get to the Sun-Earth L_2 point approximately 1.5 million km from the Earth (away from the sun) and for L_2 station-keeping.
Delta-H (DH)	Uses IRUs and the PCS to perform momentum unloading.

MATRIXx Tool Set



- System Parameters (Xmath %VARs) are defined in Xmath and exported to System Build. These are used for FSW tables and commands.

SystemBuild Environment

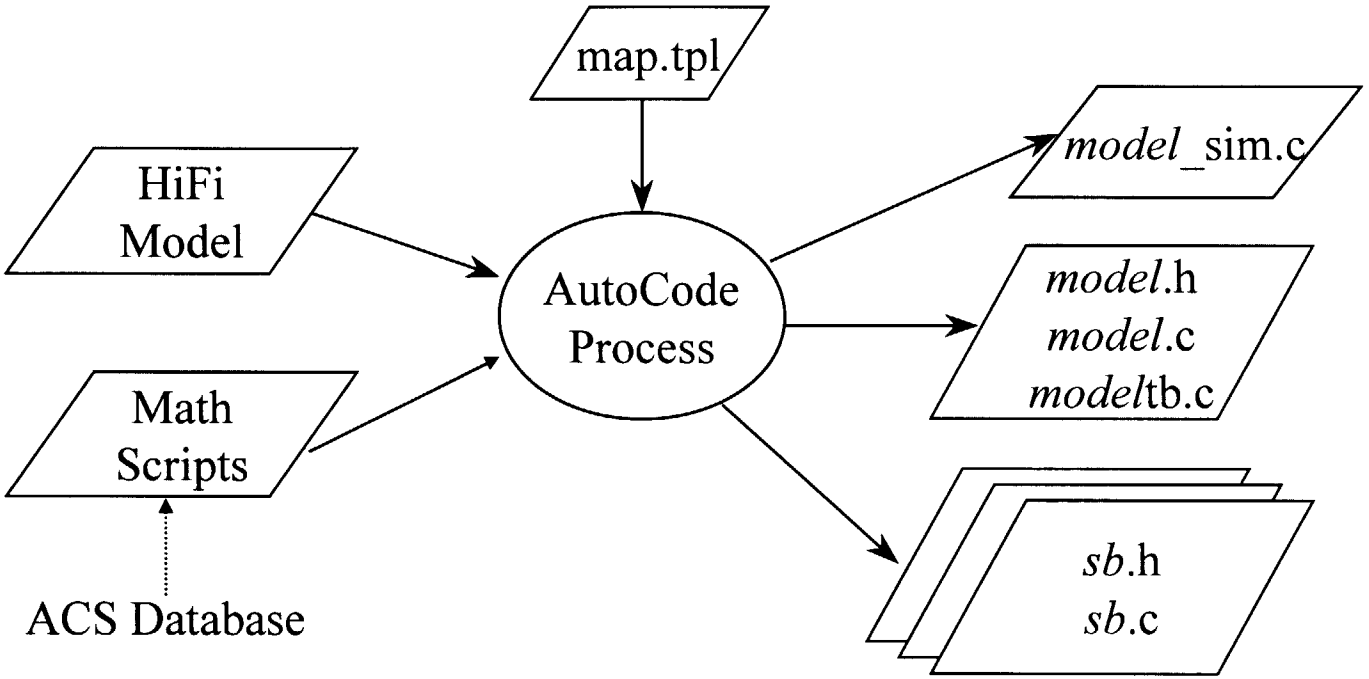
- SuperBlocks are hierarchical objects
  - Provide logical structure
  - Contain other blocks
- SuperBlock timing attributes
  - *Discrete*, Continuous, *Procedure*, Triggered
- Procedure SuperBlock types
  - *Standard*, *inline*, macro, interrupt, background, or startup

Italicized SuperBlock attributes are being used by MAP for AutoCode

SystemBuild Environment

- Common functional blocks
  - Trig functions, algebraic functions, and dynamic systems functional blocks
- "Open" blocks include
  - Algebraic Blocks
    - Define outputs as the result of algebraic functions of the inputs and block parameters
  - BlockScript
    - Limited structural programming environment using FORTRAN-like language
  - User Code Blocks (UCB)
    - Import user written code

Code Generation Process

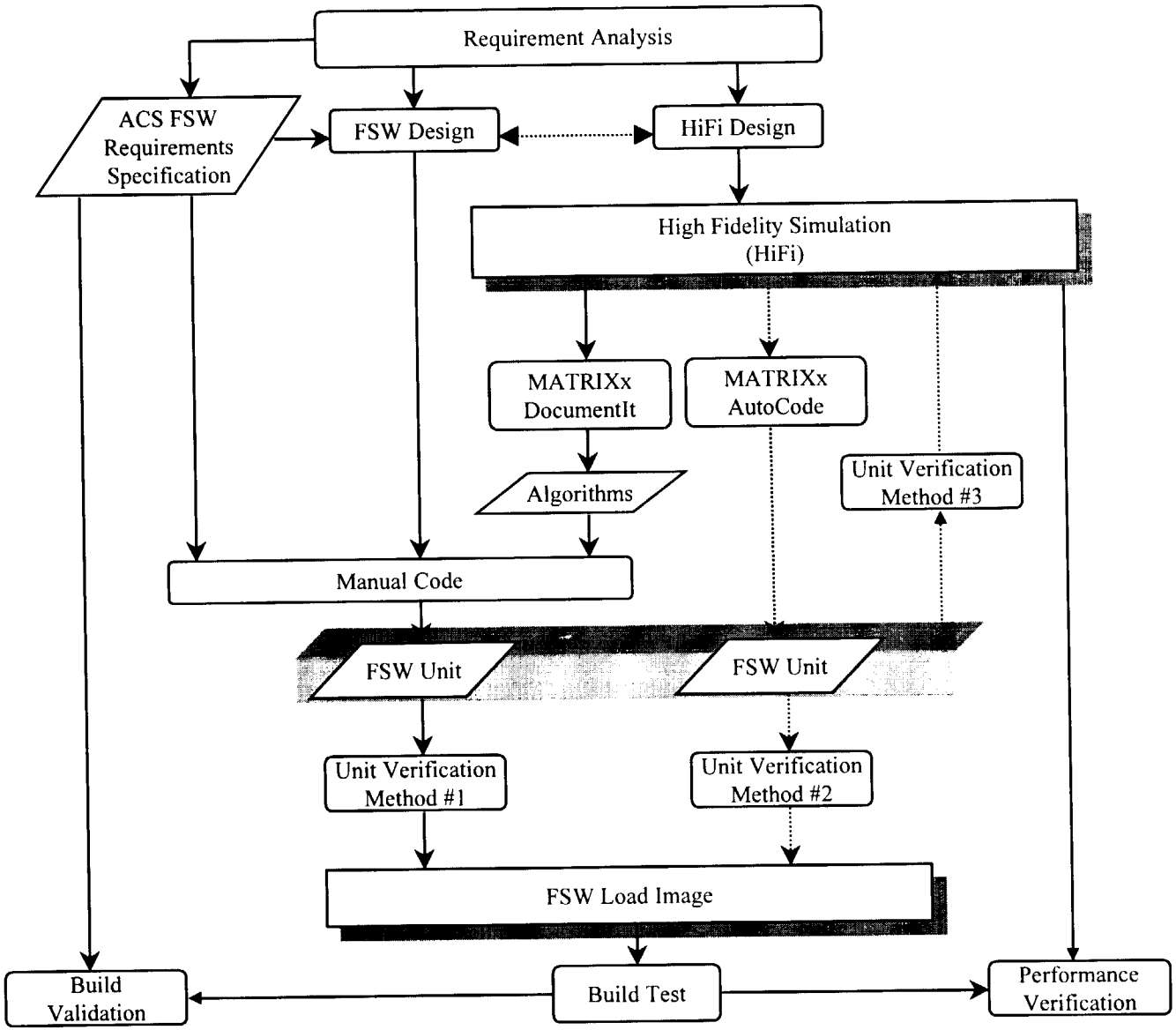


- *model* is the model name supplied to AutoCode. For map "achifi" is used.
- *sb* is the SuperBlock name defined in the HiFi.
- *model*_sim.c contains any non-FSW code. E.G. UCB wrappers.
- Process
  - Load HiFi model
  - Execute MathScript files to define Xmath variables
  - Run AutoCode to generate the code

Code Generation Template (TPL) File

- Extended ISI's TPL file (renamed to map.tpl) to
  - Create separate header file and source file for each SuperBlock
  - Create *model*_Init() and *model*_Dispatch() functions to clarify interface and provide placeholder for customized code
- "Closed" automatic code generation
  - ISI supplied TPL libraries generate the code
  - @declare_percentvars()@ generates all of the %VAR declarations

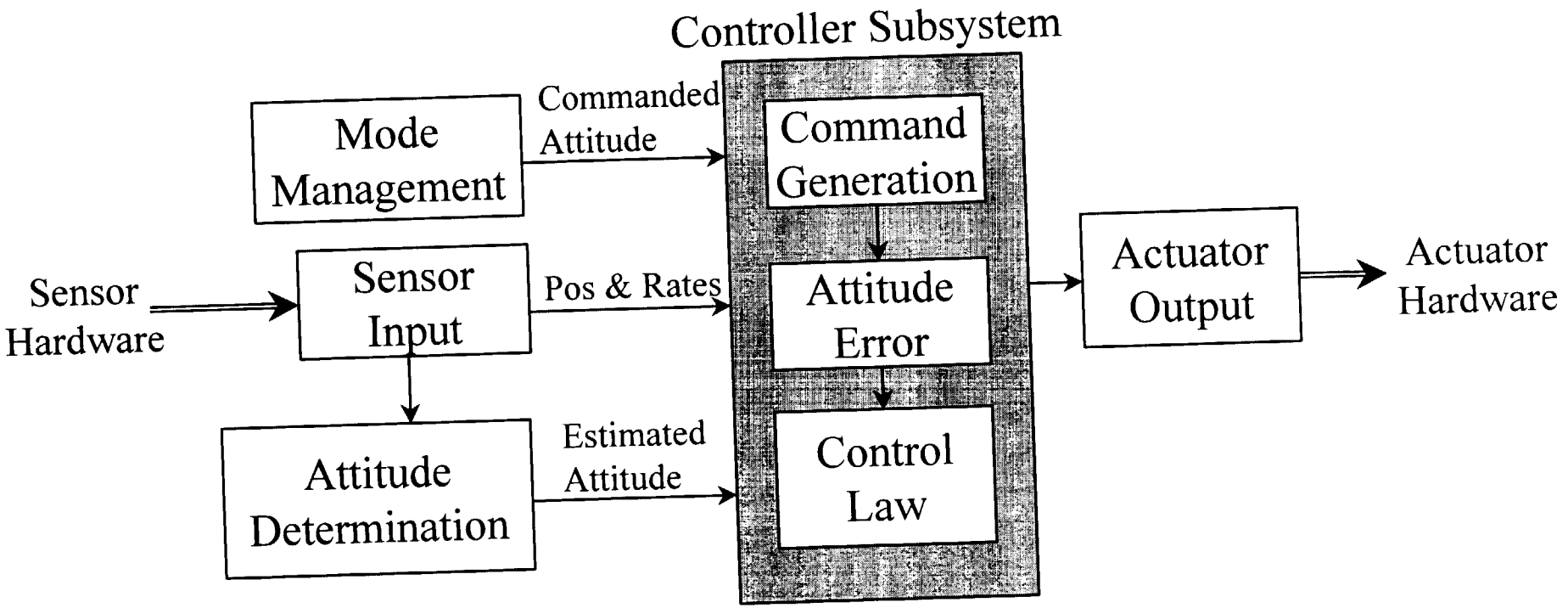
Software Development Process



Forces Driving the Automatic Code Scope

- Minimize risks and maintain schedule
- High algorithm-to-code ratio
- Low HiFi-to-FSW architectural coupling
  - No ground command handlers
  - No software bus interfaces
  - No FDC notification, or asynchronous event message generation

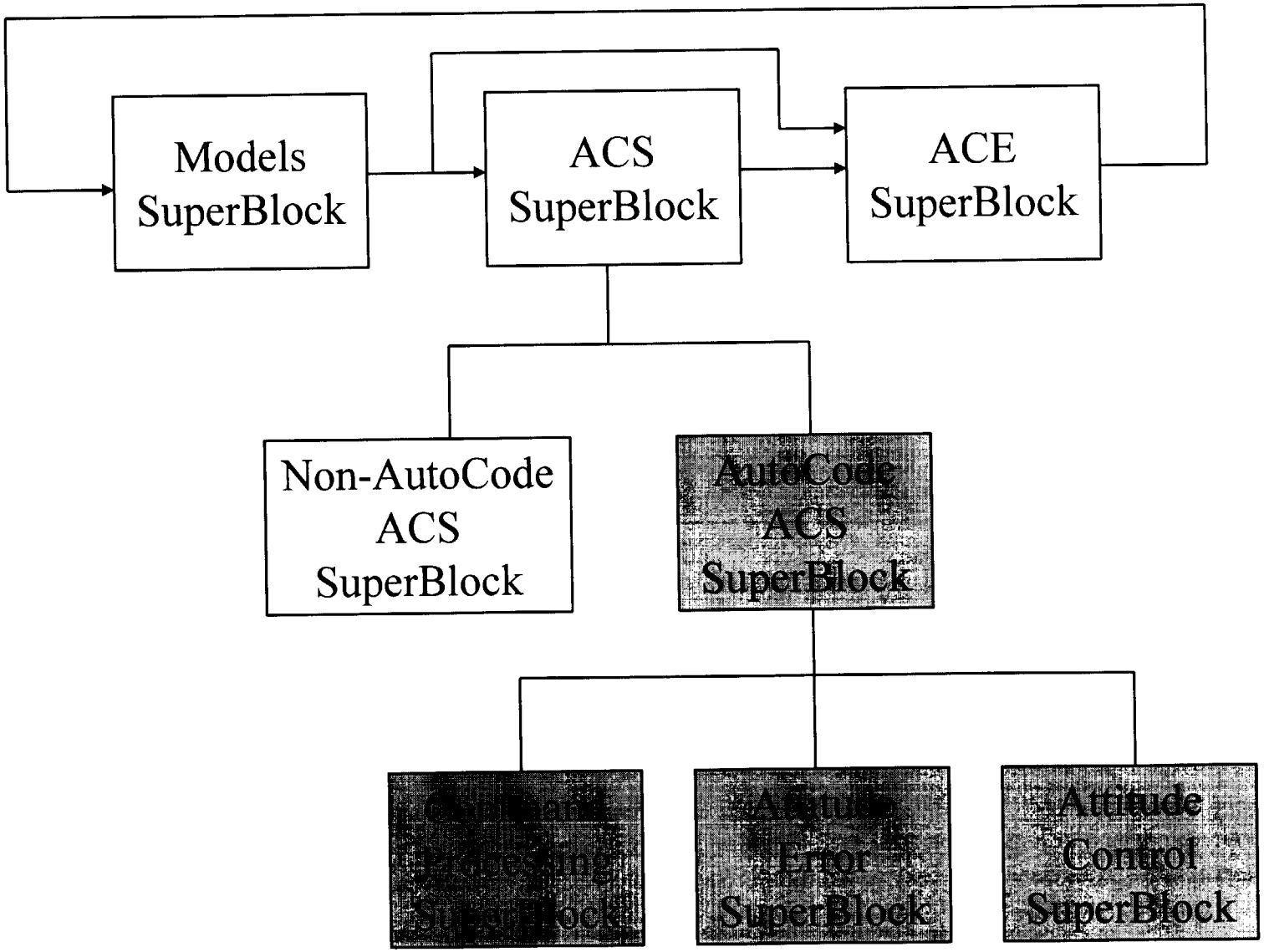
Defining the Automatic Code Scope



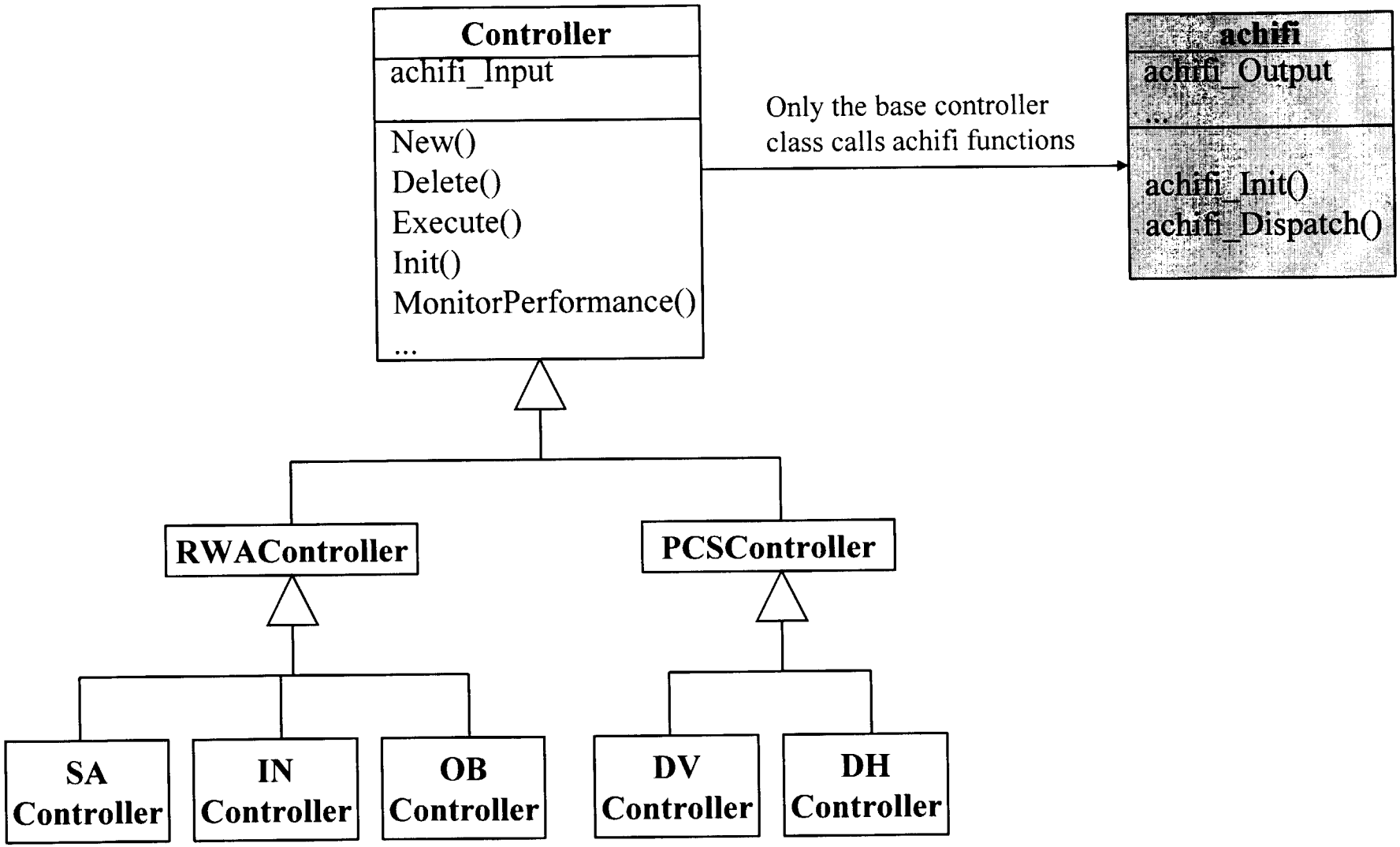
Consequences of the Design

- FSW requirements for HiFi AutoCode SuperBlock
  - FSW commanded and computed values supplied as inputs to top-level procedure SuperBlock
  - FSW telemetry and onboard computation needs define required outputs from top-level procedure SuperBlock
  - Any non-commanded ground modifiable parameter must be defined as a %VAR for inclusion into a FSW table
- Single rate system so no need to use AutoCode's scheduled-subsystem option
- Relatively simple HiFi interface to be managed by manual FSW

HiFi Implementation



Manual Flight Code Implementation



Manual Flight Code Evaluation

- Object-Oriented Design (OOD) implemented in C
- AutoCode dependencies encapsulated by base controller class
- achifi viewed as a single object
  - Two member functions: achifi_Init() and achifi_Dispatch()
  - achifi_Output treated as read-only output by other FSW subsystems
- OO controllers resulted in small easy to test functions

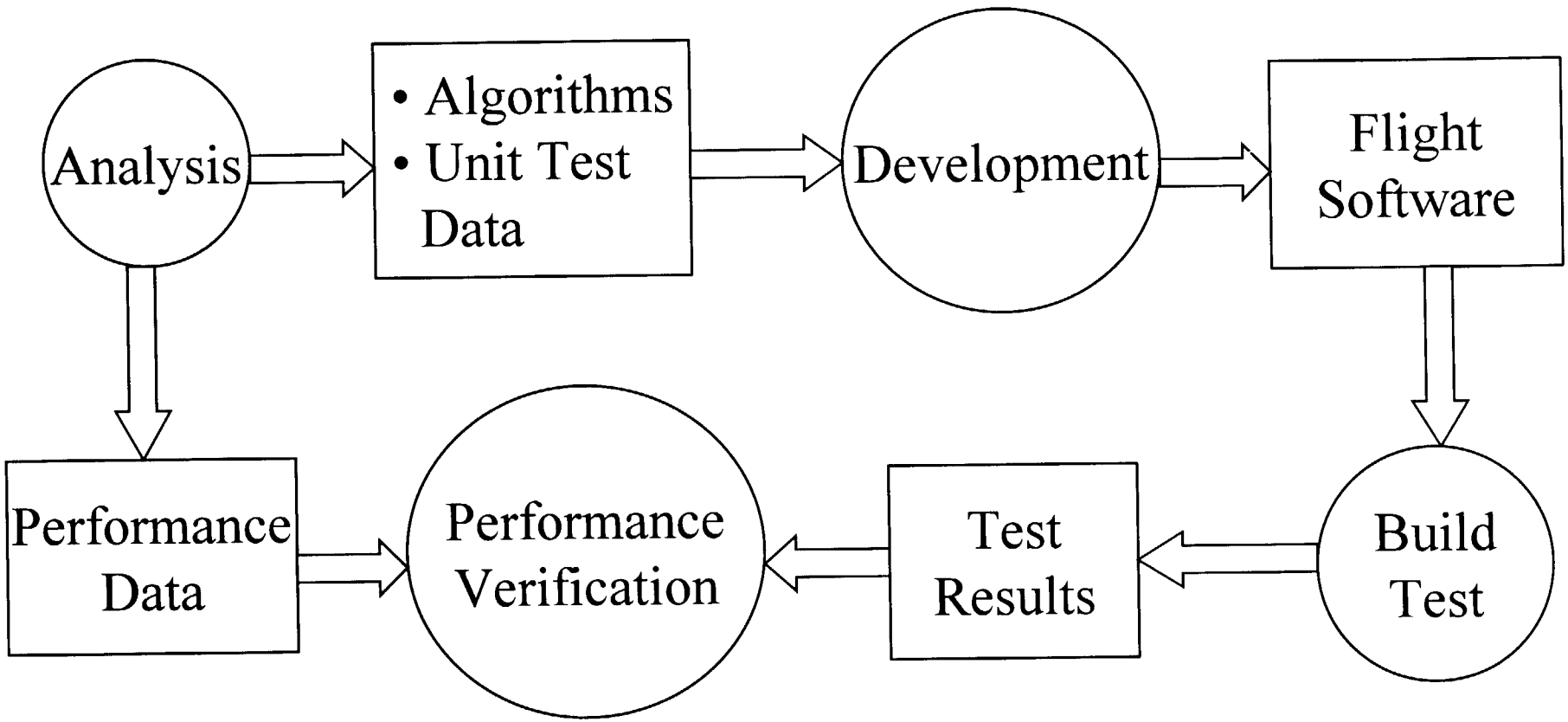
Automatic Code Evaluation

- Non-ANSI C function prototypes
  - Compiler warnings a nuisance, but no errors due to non-ANSI compliance
- Non-inline SuperBlocks result in code about twice as long as equivalent manual code. Inefficient but . . .
  - Team opted for separate files for each function (couldn't use inline attribute) for easier FSW configuration and maintenance
  - MAP has plenty of CPU and memory resources
  - Manageable file sizes and code is relatively easy to follow
- Inline comments reference SystemBuild block names/ID for traceability

Automatic Code Evaluation

- A 1 second delta-time is hard coded
  - Manually modified generated code to use computed delta time
- Two SuperBlocks did not result in variables being used for %VARs
  - Designed workarounds in HiFi; No manual code changes
- Flight tables require contiguous data, but Xmath %VAR partitions do not translate to C structures
  - Output %VARs to a separate file and used linker scripts to ensure contiguous data storage; No manual code changes
- Team has verified automated code correctly implements the HiFi design

Lessons Learned - Process Improvements



Lessons Learned - Process Improvements

- Improved configuration management
  - Synchronized HiFi and FSW builds
  - Synchronized HiFi and FSW parameter definitions
- Improved communication
  - Analysts integrated into all phases of FSW
  - Adopted naming conventions for HiFi and FSW
  - MathScripts (next slide) documented HiFi tests
- Improved unit testing
  - Similar HiFi and FSW designs enabled better test data flow

Process Improvements

- Improved automation
  - MathScript (Xmath command files) driven HiFi simulation
  - Graphical menus for easy plot manipulation
  - Automated build test plot generation
- Consistent and efficient data analysis
- Improved performance verification
  - Generation of MathScript files from build test procedures and test data
- Many process improvements were the result of the entire tool set, not just AutoCode

Measuring Process Improvement

Lines Of Code(LOC)	Spacecraft	Man Years (MY)	LOC/MY
33,318	XTE	13.8	2414
17,525	MAP	6.1	2872

- Indicates MAP has been more productive but…
  - Metrics are limited to total developer time charged to a project; not activity specific
  - No metrics for analysts or build testers
  - MAP production rate is inflated
    - Automatic code is less efficient and 5,356 of MAP LOC (31%) are automatically generated

DETERMINING SOFTWARE (SAFETY) LEVELS FOR SAFETY-CRITICAL SYSTEMS

Doris Y. Tamanaha
dtamanaha@west.raytheon.com

Meng-Lai Yin
mlyin@west.raytheon.com

Raytheon Systems Company
Loc. FU, Bldg. 675, M/S AA341
1801 Hughes Drive, Fullerton, CA 92834

ABSTRACT

For safety-critical software-intensive systems, software (safety) levels are determined so that the appropriate development process is applied. This paper discusses issues of applying the results of fault tree analysis to software (safety) levels determination. In particular, the inconsistency problem, i.e., inconsistent software (safety) levels, is addressed and an approach is presented.

Keywords: Fault tree analysis application, safety-critical systems, software (safety) levels.

1. INTRODUCTION

For safety-critical systems, process requirements to develop the software need to be met. Several standards have been evolved which classify processes into levels, such as the RTCA/DO-178B "Software Considerations in Airborne Systems and Equipment Certification"[1] or the Software Engineering Institute Capability Maturity Model [2][3][4]. Applying the results of fault tree analysis to determine the software (safety) levels is proposed in this paper, since fault tree analysis has been widely used for safety-critical systems.

For large stringent systems, inconsistent software (safety) levels can occur. This is due to the various concurrent activities of different organizations, e.g., the software development group, the system architecture group, and the safety group. This paper describes the inconsistency problem, and the strategy and methods to deal with this problem. The goal is to ensure that appropriate software (safety) levels are applied to the developed software.

2. DETERMINING LEVELS

2.1 Software (Safety) Levels

Software (safety) levels determine the associated process to be followed by the software developers. There are several existing development processes defined based on software levels, such as the Capability Maturity Model (CMM) by the Software Engineering Institute (SEI), the ISO 9000 series of standards by the International Organization for Standardization [2][3][4], and the RTCA/DO-178B [1]. The discussion here focuses on the RTCA/DO-178B standard.

The RTCA/DO-178B "Software Considerations in Airborne Systems and Equipment Certification" [1] provides guidelines for the production of software for airborne systems and equipment [5]. In particular, five categories are identified for the failure conditions, i.e., catastrophic, hazardous, major, minor, and no effect. Five software (safety) levels are defined accordingly, i.e., level A, B, C, D and E. The software (safety) level determines the development effort that demonstrates compliance with certification requirements.

2.2 General Rules

The top-down methodology based on the fault tree models is fairly straightforward. The fault tree considers not only the events related to the software, but all the possible events that can cause the top event (root event). The methodology first determines the safety level of the top event, then follows the 2 general rules listed below: (1) For the events under an OR gate: the safety level of these events are the same as that of the top event of the

OR gate. (2) For the events under an AND gate, three cases are distinguished: (2a) If the event is associated with a monitoring function, i.e., that it monitors some other function(s), then this event has the same level as that of the top event of the AND gate. (2b) If the event is associated with a monitored function, i.e., its function is monitored by some monitoring function, then it can have a level lower than that of the top of the AND gate. The philosophy is that we believe the failure of this function can be detected and corrected by the monitoring function. (2c) If the events under an AND gate do not have the monitoring/monitored relationship, then they will inherit the same level as that of the top event of the AND gate. However, if these events are truly independent, then a level lower than the top event can be assigned. For the example shown in Figure 1, assuming the effect of the top event (Hazardously Misleading Information) is classified as level B according to RTCA/DO-178B. Then, the safety level for this HMI is B. If IE1 and IE2 are functions that have the monitored/monitoring relationship, e.g., IE1 is the monitored function and IE2 is the monitoring function, then IE1 has safety level D and IE2 has safety level B.

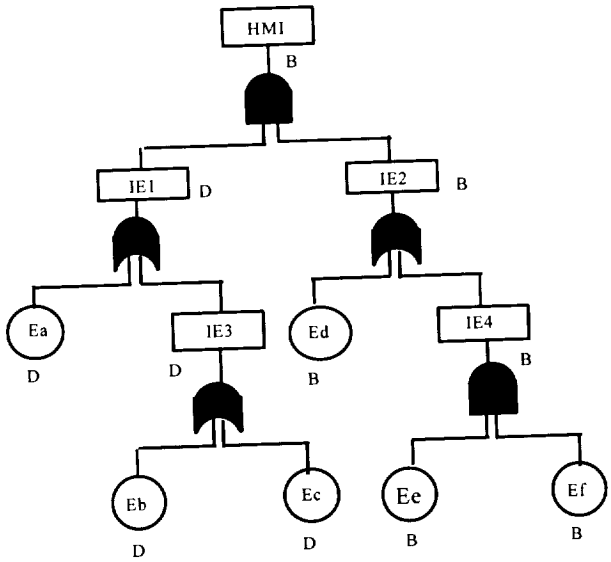


FIGURE 1. EXAMPLE FAULT TREE

For the basic event Ea and the intermediate event IE3, since IE1 has safety level D, they are marked as level D in accordance with rule 1 above. Moreover, for IE4, assuming the basic events Ee and Ef under the AND gate do not have the monitored/monitoring relationship. Thus, they are both marked as level B. For efficiency, minimum cut sets can be used as assistance. Moreover, some engineering judgement is necessary when conflicts occurred [6].

2.3 The Process

A six-step process relates the fault tree analyses to the software activities is presented in Figure 2. The first three steps are the preliminary marking, whose results are recorded in a database called the Requirement Management System (RMS). The subsystem fault tree analyses and data flow analyses were performed as parts of the preliminary marking. The safety group conducted fault tree analyses, while the software people conducted the data flow analyses. The subsystem fault trees identify *software capabilities* that can cause a hazard. In other words, if a failure of a software capability contributes to a hazard, it is identified in the subsystem fault tree. Thus, the safety level for the software capability can be marked, based on the general rules described above. The marking results need to be integrated into the software development process. The model used for data flow analyses, referred to as the capability model, is marked for this purpose. Finally, the results are recorded into the RMS database.

When the preliminary marking is finished, the software development process moves to the stages of preliminary design and detailed design. It is during this time frame that the software level marking is refined and updated through extending the subsystem fault trees. Extending the subsystem fault trees is based on the information provided by the software preliminary design and detailed design. In software preliminary design, the CSCs (Computer Software Components) of each CSCI (Computer Software Configuration Item) are defined, as are the major data stores and interfaces among CSCs. Thus, the original subsystem fault tree can be extended to the CSC level. In the detailed design phase, the processes are further decomposed into Computer Software Units (CSUs) and functions. Hence, we can extend the fault trees to the CSU level.

Due to the characteristics of large systems that are composed of several organizations with different objectives, inconsistent software (safety) levels are expected. The results of this inconsistency are schedule costs and safety risks. Hence, the

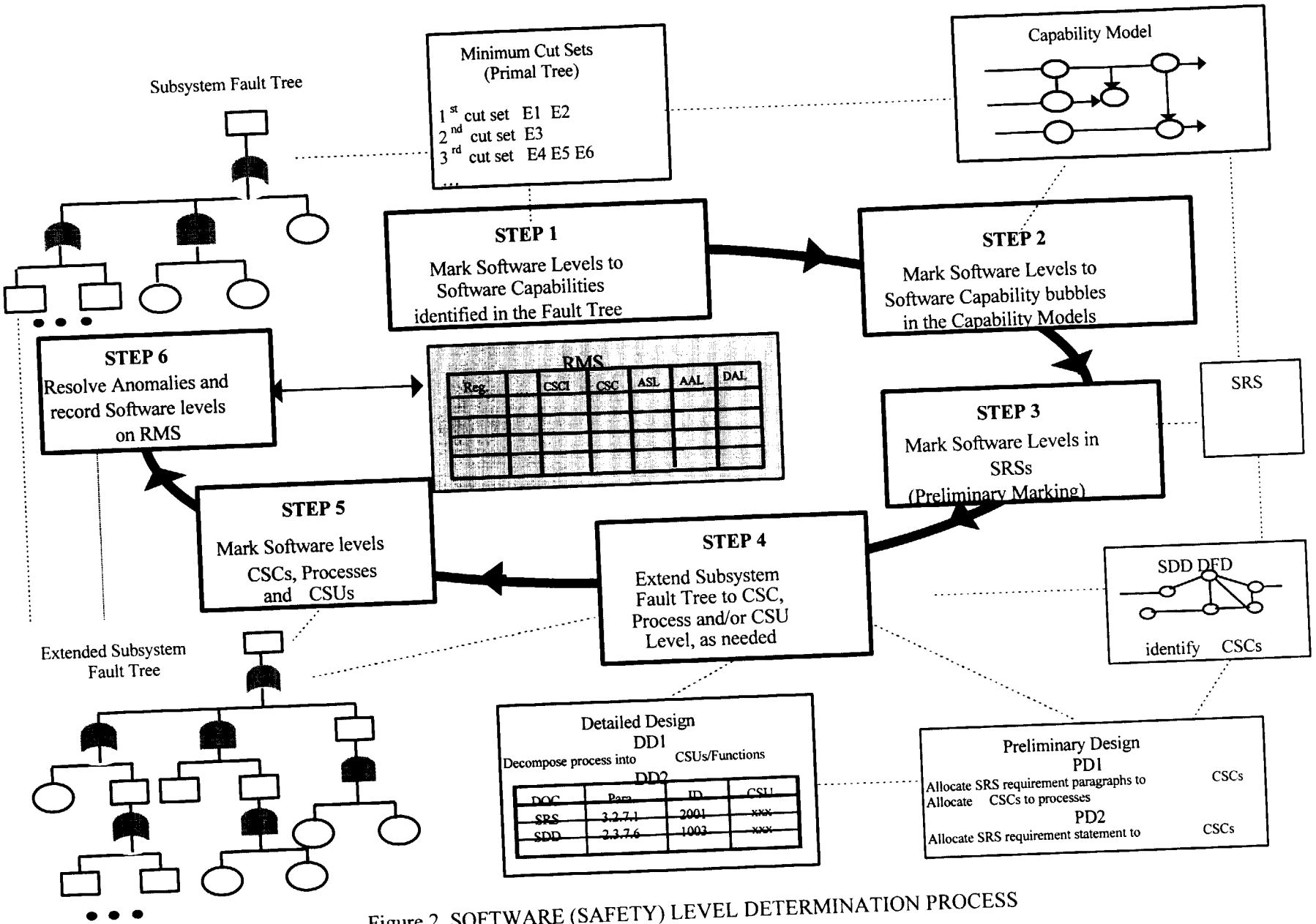


Figure 2. SOFTWARE (SAFETY) LEVEL DETERMINATION PROCESS

inconsistency needs to be resolved so that one accepted development process can be applied. The inconsistency problem is discussed next.

3. THE INCONSISTENCY PROBLEM

3.1 Inconsistent Views

Large programs entail several organizations that have impact on the software (safety) levels. Unfortunately, these organizations often have different views and responsibilities, which may conflict with each other, often as a result of changing requirements that affect revisions at multiple levels, e.g., within the architecture, software, or safety constraints. Hence, resynchronization is needed.

The three organizations that are related to the software (safety) levels are the software development, system architecture, and safety groups, as shown in Figure 3. The software group follows a process to develop the software. System architecture (with software representation) partitions the system and decides which software resides on which platforms. Usually, a single, consistent process is followed for the software developed on the same platform. The safety group analyzes the system and derives software (safety) levels as requirements.

If the problem of inconsistent software (safety) levels is not resolved, the software may not be developed appropriately. If the inconsistency is not resolved in a timely manner, schedule will be slipped, and cost will be increased. Moreover, if the inconsistency is not resolved correctly, the software development process can be inadequate. In short, the inconsistency problem needs to be resolved correctly and in a timely fashion in order for the system to be built.

4. THE APPROACH

4.1 The Basis

A basic philosophy we took is that the inconsistency is expected. Therefore, the existence of all the software levels shall be recorded. From there, the inconsistency can be identified. Only if we can recognize the inconsistency can the inconsistency problem be addressed and resolved.

There are two types of inconsistencies. The first type is referred to as the tolerable inconsistency where the software developers follow a process that exceeds the current process requirements. This tolerable inconsistency implies that developed software can be used in later phases when a higher level is required (software levels are interpreted as A > B > C > D > E, e.g., level A is higher than level B, etc.) The second type of inconsistency is intolerable, where the process that the developers follow does not meet the current process requirements for safety certification. Intolerable inconsistencies must be identified and resolved.

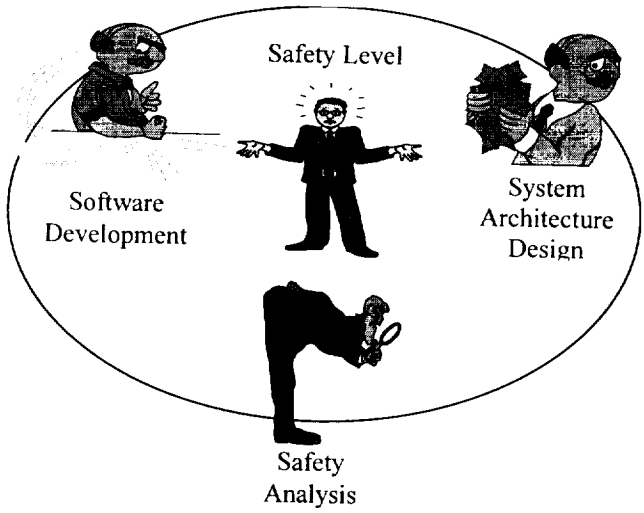


FIGURE 3. DIFFERENT VIEWS OF SOFTWARE (SAFETY) LEVELS

4.2 The Process of Managing Inconsistent Software (Safety) Levels

To manage the inconsistent software (safety) levels, three major steps are proposed: (1) maintaining software (safety) levels associated with different organizations, (2) cross-checking the recorded software (safety) levels and identifying the inconsistency areas, and (3) resolving the inconsistency by involved organizations and engineers.

A process is defined to manage the inconsistency problem, as shown in Figure 4. The process is iterative, since the system development itself is iterative. A central repository, e.g., the Requirement Management System database, is maintained as which is the center of the process. Configuration

control of the database is necessary to ensure the overall consistency of the software (safety) levels. Cross checking different software (safety) levels will identify the inconsistent areas. To resolve the problem of inconsistency involves the teamwork efforts of software engineers, system engineers and safety engineers.

4.3 Maintaining the Software (Safety) levels

Because inconsistency is expected, all the differences can be managed. The strategy is to record all the software levels resulting from different organizations, recognize any inconsistency, and deal with the intolerable inconsistency problems. The different software (safety) levels (due to the various organizations) are recorded in the Requirements Management System database. Three different software (safety) levels are maintained, the "Assigned Software Level" (ASL), the "Assessed Architecture Level" (AAL), and the "Development Assurance Level" (DAL). A detailed description of each of the levels is addressed below. The goal of managing the inconsistency is to assure that ASL ≤ AAL ≤ DAL (software levels are interpreted as A > B > C > D > E, i.e., level A is higher than level B, etc.).

The ASL focuses on the severity effects and hazard mitigation. The ASL is assigned based on the system safety assessment results, e.g., the fault trees, as described in Section 2. Note that this ASL serves as the minimum acceptable software (safety) level, as it is based solely upon the referenced safety analysis and functional mitigation. The ASL is implementation independent. The AAL is used to reflect design constraints from architectural allocation of software capabilities. To prevent software developed at a low level process from corrupting software developed at a higher level process, several mechanisms are considered, such as the firewall concept. However, to reduce the complexity of the inconsistent process throughout the whole system, a "safety system high" concept is used. The AAL is determined based on the *highest* severity level of software assigned to that platform. For example, to simplify the development process management, all software developed on a platform certifiable to level B is developed to level B, even if the software has an assigned safety level or ASL of D.

The incremental strategy has been widely used for large safety-critical systems. Not only because a program needs to improve as the equipment and technology improve, but also because the safety concern is changing as the phases proceed and the system becomes operational in a production sense. To prevent rework efforts as much as possible, the DAL can be used to demonstrate the achievement of compliance to final phase requirements. This DAL is committed to by software development and is a development strategy to meet or surpass the current AAL requirement. To accommodate the incremental strategy, a separate set of the three software (safety) levels is maintained. An advantage of maintaining different software (safety) levels is that it helps an individual organization to focus on its own tasks. In particular, the software development team only needs to focus on the DAL and develops the software to the assigned DAL. It is the safety engineers responsibility to assure the relationship of ASL ≤ AAL ≤ DAL.

5. RESULTS

5.1 Identified Anomalies

Anomalies are the intolerable inconsistencies. This section describes the anomalies that are identified during the implementation of the process, and a general process of how to resolve these anomalies. Two types of anomalies are recognized, e.g., internal and external. The internal anomalies are due to the sharing of the same software requirement by different software configuration items. For example, a system service function may be used in several subsystems that are on different platforms. Different failure effects may be estimated, since different safety concerns are applied to different subsystems. As a result, different ASLs are assigned for the same software requirement. The internal anomalies can be resolved by assigning the *highest* software (safety) level to the software being concerned. In other cases, similar design may be used on platforms at different certification levels. In those cases, the safety level for the requirements allocated to the similar design carries a dual designation, say "B/D". It is then understood that

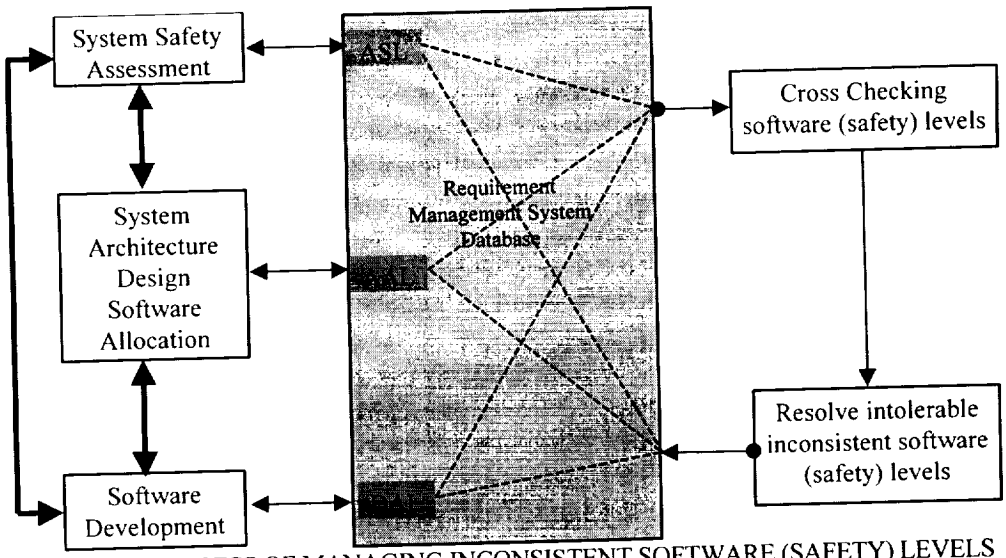


FIGURE 4. PROCESS OF MANAGING INCONSISTENT SOFTWARE (SAFETY) LEVELS

those requirements must be examined for two processes.

External anomalies are caused due to the inconsistency. The identified external anomalies are: (1) The DAL for a particular entry in the database is denoted as N/A (non applicable) or the value is missing, while the corresponding ASL (and/or AAL) has a level assigned. This occurs especially when COTS (Commercial Off The Shelf) products are used. (2) One of the software (safety) levels (AAL, or DAL) has two values assigned, while the other one has only one level. (3) The DAL is lower than the AAL and ASL.

5.2 Resolving Anomalies

The safety engineers, software development team leads, and system engineers are informed of the anomalies that have occurred. For each anomaly identified, corresponding safety engineers and software engineers work together to resolve the problem. Once the anomalies are resolved and a consensus is reached, a "Software Change Control Board" reviews and approves the request for level changes. This satisfies the issues of configuration control for the database. Software safety engineers assign the ASL and AAL changes that are reviewed internally by the software safety team. Moreover, software safety engineers participate on the Software Change Control Board with sign-off capability for all three software levels, i.e., ASL, AAL, and DAL. Recall that the goal of this process is to assure that ASL ≤ AAL ≤ DAL.

6. CONCLUSION

In this paper, we present a method of determining software (safety) levels based on fault tree analysis. The inconsistency problem resulting from the need to operate concurrent activities to meet schedules in building large, complex systems is addressed and a strategy of handling it is discussed. The software levels determined using this approach demonstrate the safety quality of a safety-critical system. Moreover, the approach is suitable for systems developed incrementally. Extensions being investigated are the relationship of software (safety) levels to other analysis approaches, e.g., safety-critical thread analysis and the use of software fault-injection techniques to harden the software itself, guided by the software level markings.

7. REFERENCES

[1] *Software Considerations in Airborne Systems and Equipment Certification,* Document No. RTCA/DO-178B, prepared by Special Committee 167 of RTCA, December 1, 1992.

[2] *A System Engineering Capability Maturity Model,* Version 1.1, System Engineering Capability Maturity Model Project, Carnegie Mellon University Software Engineering Institute, SECMM-95-01 CMU/SEI-95-MM-003, Nov. 1995.

[3] *The Capability Maturity Model: Guidelines for Improvement the Software Process,* Carnegie Mellon University, Software Engineering Institute, Addison-Wesley Publishing Company, 1995.

[4] Mark C. Paulk, "How ISO 9001 Compares with the CMM", IEEE Software, January 1995.
[5] *Global Position System: Theory and Applications*, Volume I and II, American Institute of Aeronautics and Aeronautics, Inc. 1996.
[6] G. Watt, "Phase 1 Software Level Marking Guidelines", WAAS SEN 5-2-5, 1997.

Raytheon

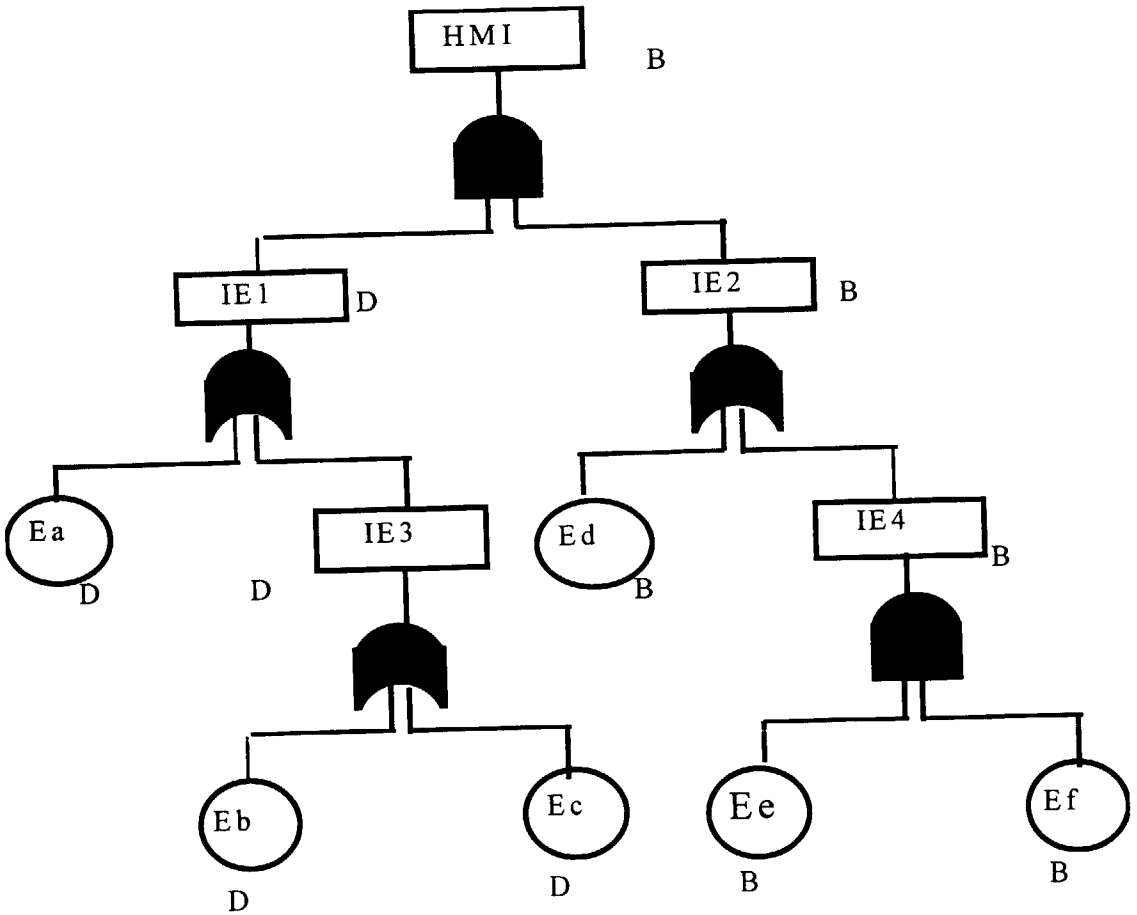
DETERMINING SOFTWARE (SAFETY) LEVELS FOR SAFETY-CRITICAL SYSTEMS

Doris Tamanaha
714-446-3050

Meng-Lai Yin (Presenter)
714-446-4269

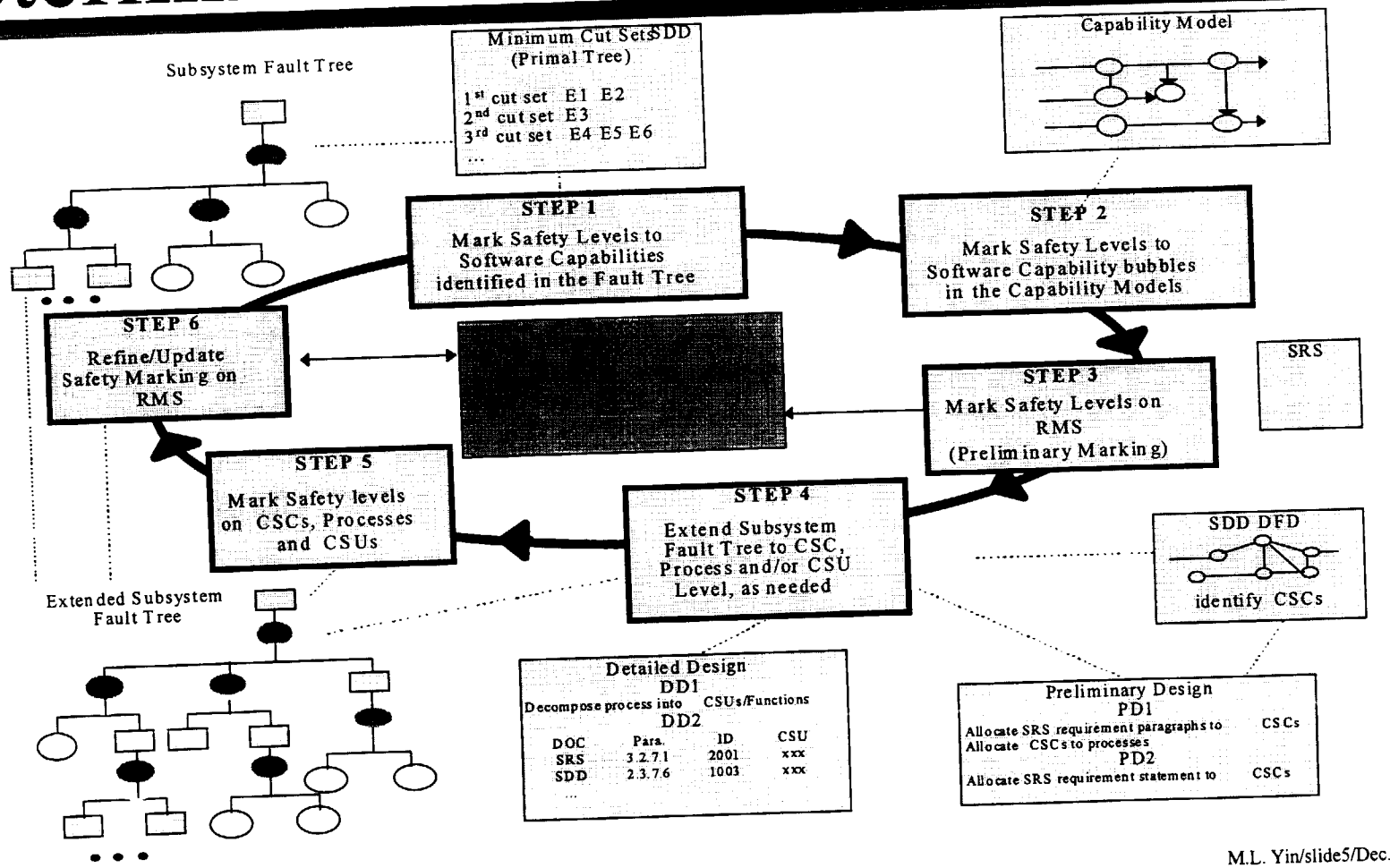
Raytheon

Example Fault Tree



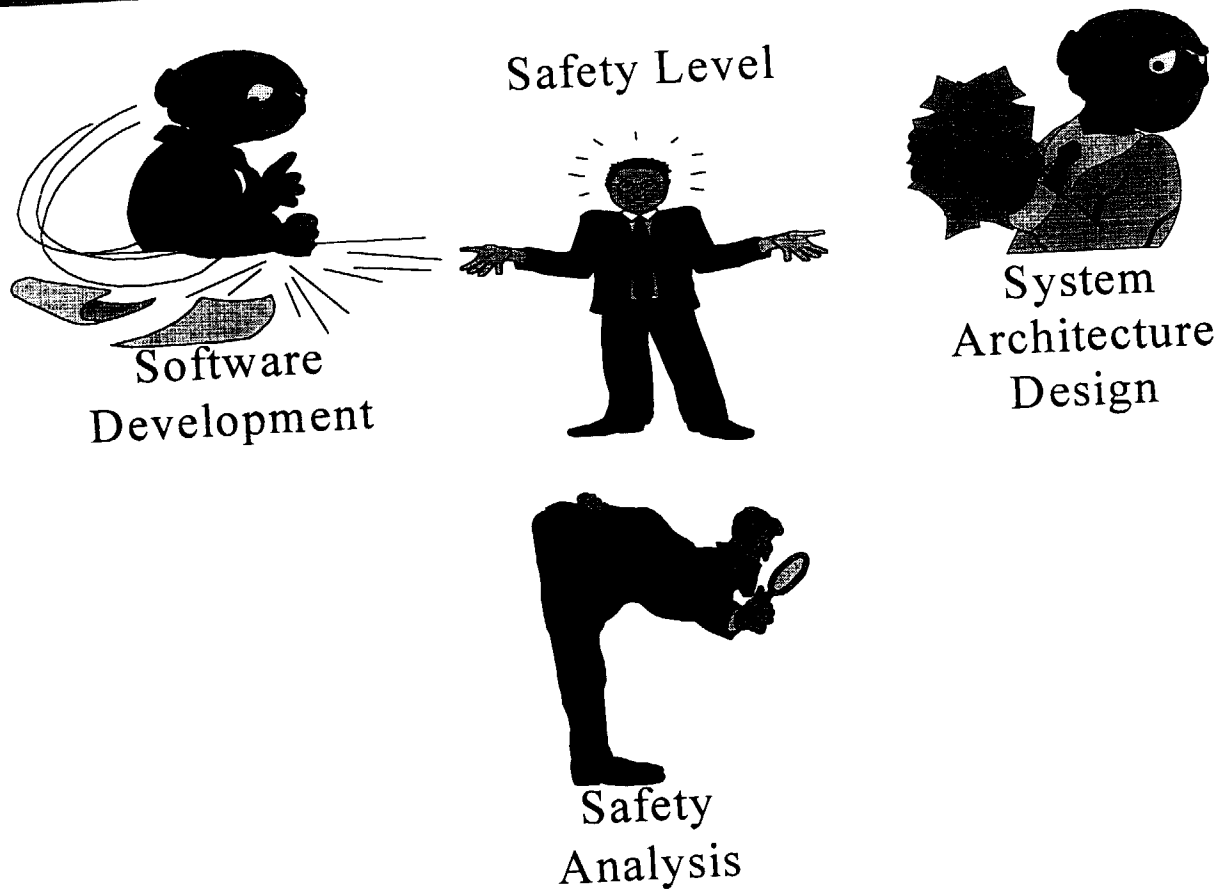
Software (Safety) Level Determination Marking Process

Raytheon



Different Views of Software (Safety) Levels

Raytheon



Managing Inconsistency (1)

- Basic philosophy: inconsistency is expected

Thus,

- record all existing software levels
- from this, the inconsistency can be identified
- only if we can recognize the inconsistency can the inconsistency problem be addressed and resolved

Managing Inconsistency (2)

STEP 1.

- maintain different software (safety) levels associated with different organizations (RMS database)
- assure ASL ≤ AAL ≤ DAL

  ASL: Assigned Software Level

  AAL: Assessed Architecture Level

  DAL: Development Assurance Level

Managing Inconsistency (3)

STEP 2.

- Cross-checking the recorded software (safety) levels and identifying the inconsistency areas

STEP 3.

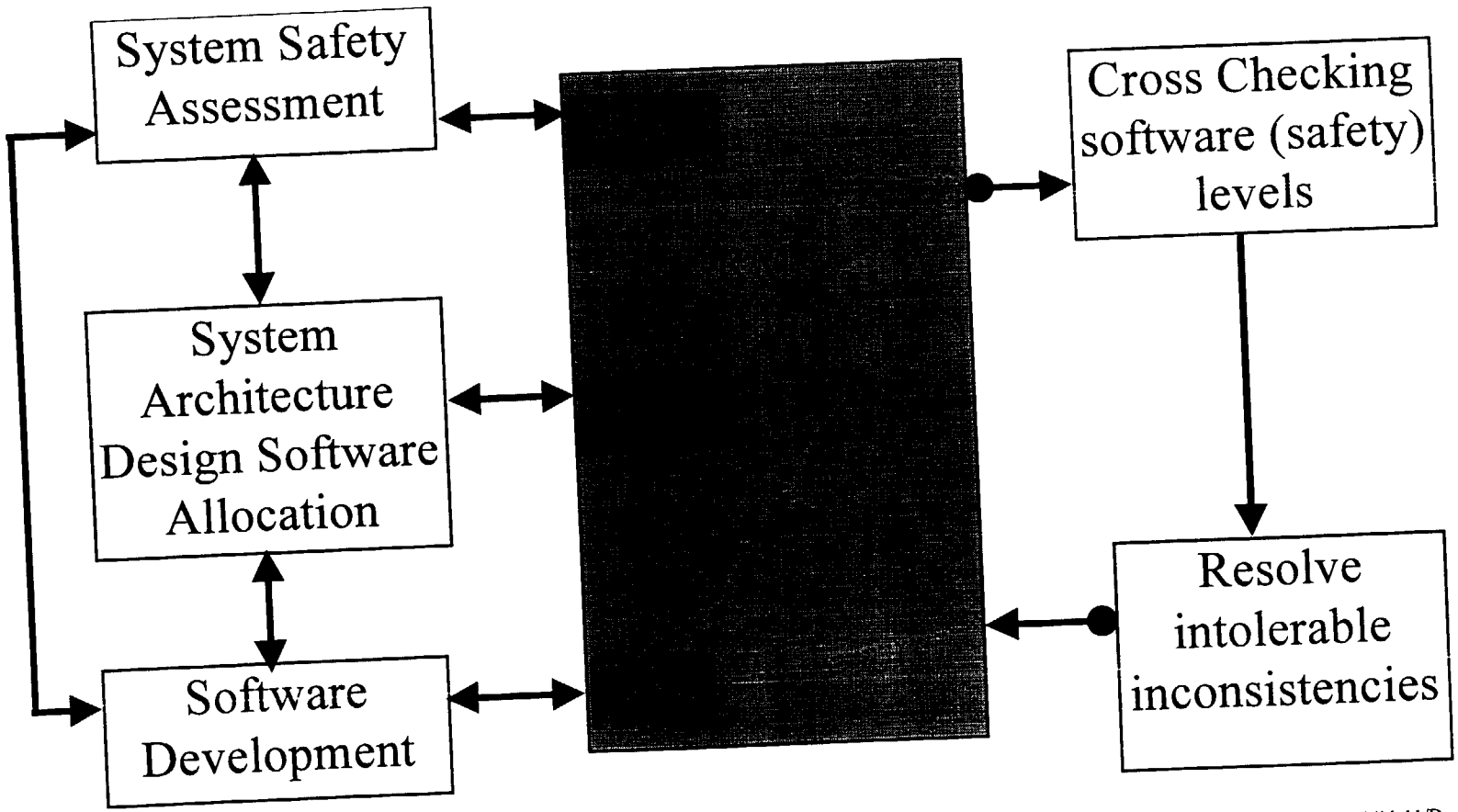
- Resolving the inconsistency by involved organizations and engineers

Types of Inconsistency

- tolerable: software developers follow a process that exceeds the current process requirements
- intolerable: software developers follow a process that does not meet the current process requirements (for safety certification)

Process for Managing Inconsistency

Raytheon



Raytheon

Implementation Results (1)

- 2 types of anomalies: internal and external
- internal (case 1)
  - due to the sharing of the same software by different configuration items
  - different ASLs are assigned for the same software
  - resolved by assigning the highest level

Implementation Results (2)

Raytheon

- internal (case 2):
  - similar design used on different platforms (the platforms have different levels)
  - dual designation (i.e., B/D) are carried for the same software requirements
  - resolved by examining the requirements for two processes

Implementation Results (3)

Raytheon

- External anomalies
  - DAL is N/A, while ASL and/or AAL has a level assigned (example, COTS products)
  - One of the levels has two values assigned, while the others has only one level
  - The DAL is lower than the AAL and ASL
  - * Resolving external anomalies involves corresponding safety engineers, software development team leads, and system engineers (assure ASL ≤ AAL ≤ DAL)

Summary

- The method and the process for Marking Software (safety) Levels
- The Inconsistency Problem
- The philosophy and the method of handling the inconsistency problem
- The implementation results

Appendix A – Workshop Attendees

Abshire, Gerald
Computer Sciences Corp.
Addy, Edward A.,
NASA/WVU
Agresti, Bill W.,
MITRETEK Systems
Allen, Guy,
OAO Corp.
Alves, Heidi,
Dept. of Commerce
Anderson, Barbara,.
Jet Propulsion Lab
Andolsek, Timothy G.,
Raytheon
Aves, Heidi,
[No Organization
Registered]
Ayers, Everett,
Ayers Associates

Bae, Youn Y.,
NASA/GSFC
Baer, David R.,
NASA/GSFC
Balzy, Louis J.,
NASA/ Ames Research
Center
Basili, Victor R.,
University of Maryland
Beall, Shelly,
Social Security
Administration
Becker, Greg,
Quality Systems
Solutions
Bert, Colvin,
OAO Corp.
Bhatia, Kiran,
MITRETEK Systems
Bismut, Noemie A.,
SATC
Blue, Velma D.,
DISA
Boger, Jacqueline,
Computer Sciences Corp.
Bowerman, Rebecca E.,
Pragma Systems Corp.
Brandenburg, Wilber,
SA/GSFC
Brown, Angela,
OAO Corp.
Brown, Patrick,
The MITRE Corp.
Burns, Edd,
TECHSOFT, Inc.
Butler, Sharyl A.,
NASA/JSC
Byrd, William E.,
University of Maryland -
Baltimore Co.

Callahan, John,
NASA IV&V Facility
Cannaday, Mona Lisa,
Computer Sciences Corp.
Carlson, Randall,
NSWCDD
Casadei, Alberto L.,
Westinghouse Electric
Corp.
Caulfield, Margaret I.,
NASA/GSFC
Celentano, Al,
Social Security
Administration
Chandler, Elizabeth,
NASA/GSFC
Chase, Bryant,
Social Security
Administration
Chatters, Gary,
Century Computing
Choates-Workman, Mary,
SOLIPSYS
Chu, Richard,
Lockheed Martin Corp.
Chung, John,
Computer Sciences Corp.
Collins, Miguel,
Computer Sciences Corp.
Condon, Steven E.,
Computer Sciences Corp.
Cook, John F.,
NASA/GSFC
Cooke, Robert,
NSWC
Corbin, Genie,
Social Security
Administration
Crispell, Michele,
SATC
Cuesta, Ernesto,
Computer Sciences Corp.
Cummings, Cheri,
Naval Center for Cost
Analysis

Daugherty, Marie L.,
Intermetrics, Inc.
Dawson, Jim,
Bell Atlantic
Decker, William J.,
Computer Sciences Corp.
Dhama, Harpal,
The MITRE Corp.
Dominguez, Alfonso,
Logicon Syscon
Dowen, Andrew Z.,
Jet Propulsion Lab
Dudash, Ed,
Naval Surface Warfare
Center

Eberstein, Igor,
NASA/GSFC
Eickelmann, Nancy,
NASA IV&V
Ekros, Jens-Peder,
Linkoping University-
Sweden
Elder, Matthew C.,
University of Virginia
Emery, Daniel N.,
Computer Sciences Corp.
Escobar, Francisco,
Database Platforms, Inc.
Eubank, Paul J.,
IRS

Fagan, David J.,
DCMC-Northrop
Grumman
Feather, Martin S.,
Jet Propulsion Lab
Fernandes, Vernon,
Computer Sciences Corp.
Fike, Sherri,
Ball Aerospace
Freitas, Robert L.,
NASA/GSFC
Fu, Chien-Cheng,
Computer Sciences Corp.
Futcher, Joseph M.,
Naval Surface Warfare
Center

Gardner, Michael,
Boeing
Garnett, Paul D.,
Mountain State
Information Systems, Inc.
Garrett, Don,
Dept. of Commerce/
NOAA/NWS
Gaston, Ralph,
Computer Sciences Corp.
George, Lee C.,
Lockheed Martin
Godfrey, Sally,
NASA/GSFC
Goodman, H. Alan,
Textron Systems Corp.
Gopalan, Venkat R.,
DynCorp
Gottlieb, Jordan,
SES, Inc.
Gresko, Tom,
SES, Inc.
Gross, Stephen,
The MITRE Corp.

Haddad, Maliha,
American University
Hair, Bruce,
OAO Corp.
Hammer, Theodore F.,
NASA/GSFC
Hammons, Matthew A.,
TRW, Inc.
Handler, Eugene,
TRW
Harris, Chi Cha,
DCMC-Northrop
Grumman
Harris, Jeffery,
Computer Sciences Corp.
Heasty, Richard,
Computer Sciences Corp.
Hebert, Ken,
Integrated Computer
Engineering, Inc.
Hendrick, Robert B.,
Computer Sciences Corp.
Henshaw, Clark,
NSWCDD
Heyden, Michael,
Intermetrics
Holden, James J.,
PYXIS Systems
International, Inc.
Holmes, Joseph A.,
IRS
Holt, Timothy D.,
TRW
Houchens, Connie M.,
NASA/GSFC
Hull, Larry,
NASA/GSFC
Husk, Steven M.,
Boeing Information
Services
Huy, Frank, D. N.
American, Inc.

Iskow, Lawrence,
U.S. Census Bureau
Jamison, Donald,
NASA/GSFC
Jeletic, Jim,
NASA/GSFC
Jeletic, Kellyann,
NASA/GSFC
Jepsen, Paul L.,
Jet Propulsion Lab
Jing, Yin,
Computer Sciences Corp.
Johnson, Pamela,
Naval Center for Cost
Analysis
Johnson, Pat A.,
NASA/GSFC
Jordano, Tony J.,
SAIC
Joseph, Sahji,
TRW
Juristo, Natalia,
Universidad Politecnica
de Madrid
Kassebaum, Kass,
Process & Change
Management
Kasser, Joe,
University of Maryland
Kea, Howard E.,
NASA/GSFC

Kelley, Ken,
Consultant
Kelly, John C.,
Jet Propulsion Lab
Kelly, Michael,
Computer Sciences Corp.
Kelly, William,
NASA/GSFC
Kemp Greenley, Kathryn
NASA IV&V Facility
Kester, Rush W.,
AdaSoft
Kidd, Ludie M.,
NASA/GSFC
Kierk, Isabella,
Jet Propulsion Lab
King, Theo E.,
Computer Sciences Corp.
Kirk, James A.,
American Century
Investments
Knight, John C.,
University of Virginia
Koslosky, Anne Marie,
NASA/GSFC
Kraft, Steve,
NASA/GSFC
Kuhn, Rick,
NIST
Kurs, Claire Z.,
Computer Sciences Corp.
Kutt, Peter H.,
Computer Sciences Corp.

Labossiere, Pamela,
Computer Sciences Corp.
Landis, Linda C.,
Computer Sciences Corp.
Lane, Allan C.,
[No Organization
Registered]
Larrabee, Robert C.,
ARINC/Pax River
Lawson, Carmen,
Aquas, Inc.
Leach, Ronald J.,
Howard University
Lear, Ronald D.,
Maryland Software
Industry Consortium
Levitt, David S.,
Computer Sciences Corp.
Lipsett, Bill,
IRS
Liu, Donald T.,
TRW
Lott, Christopher M.,
BELLCORE
Loving, Jr., Calvin E.,
Dyncorp
Lovisa, Robert P.,
Computer Sciences Corp.
Lowell, Kevin J.,
Raytheon ITSS/USGS
EROS Data Center
Lubelczyk, Jeffrey T.,
NASA/GSFC
Ludford, Joe,
White Hart Associates
Lue, Yvonne,
Computer Sciences Corp.
Luettgen, Gerald,
NASA/LaRC

MacKenzie, Garth R.,
University of Maryland
Manley, John H.,
University of Pittsburgh
Marciniak, John J.,
Marciniak & Associates
Mashariki, Amen,
Howard University
Maury, Jesse,
Omitron, Inc.
Maymir-Ducharme, Fred
Lockheed Martin
McClinton, Arthur,
MITRETEK Systems
McComas, David,
NASA/GSFC
McDonough, Dick,
WVHTC Foundation
McGarry, Frank E.,
Computer Sciences Corp.
Meyers, Akiko E.,
Computer Sciences Corp.
Mitchell, James P.,
TRW
Morisio, Maeurizio,
University of Maryland
Murray, Henry,
NASA/GSFC
Myers, Philip I.,
Computer Sciences Corp.

Nakano, Luis G.,
University of Virginia
Nestlerode, Howard,
Unisys Corp.
Nikora, Allen P.,
Jet Propulsion Lab
Norcio, Tony F.,
University of Maryland-
Baltimore Co.

O'Donnell, Charlie.,
ECA, Inc.
O'Mahony, Sheryl-Jean,
USAISSDCW
O'Mary, George W.,
The Boeing Company
O'Neill, Don,
Consultant
Obenza, Ray,
Software Engineering
Institute
Oliveros, Alejandro,
Universidad de la
Tiatanza
Ott, Inhwan,
Computer Sciences Corp
Pajerski, Rose,
Fraunhofer Center-
Maryland

Parizer, Michael S.,
SATC
Parra, Amy T.,
Computer Sciences Corp.
Patterson, Janine Y.,
Lockheed Martin Corp.
Pavnica, Paul,
Treasury - FinCEN
Peltier, Daryl A.,
NASA/JSC
Penix, John,
NASA/Ames Research Center
Peterson, Ivars,
Science News
Pollizzi, III, Joseph A.,
Space Telescope Science Institute
Popadiuk, Larisa C.,
SAIC

Quarles, Steve,
Computer Sciences Corp.

Ray, Keith W.,
Ki Solutions Consulting
Reid, Jon,
Computer Sciences Corp.
Reid, Mike,
Computer Sciences Corp.
Reifer, Donald,
Reifer Consultants, Inc.
Rohr, John A.,
Jet Propulsion Lab
Rosenberg, Linda H.,
SATC Unisys
Roy, Dan M.,
STP&P
Rus, Ioana,
Fraunhofer Center Maryland
Russell, Gabriella,
Dept. of Commerce

Samson, Don,
Software Process Technologies
Schleicher, Susan,
NSWC
Schneider, Frank L.,
Jet Propulsion Lab
Schulmeyer, Gordon G.,
PYXIS Systems International, Inc.
Schultz, David,
Computer Sciences Corp.
Seaman, Carolyn B.,
University of Maryland-Baltimore Co.
Sharma, Jagdish,
NOAA
Shaw, Michele A.,
OAO
Shell, Elaine,
NASA/GSFC
Shull, Forrest,
University of Maryland
Sim, Edward R.,
Loyola College
Smith, David,
SATC
Smith, Donald,
ManTech International Corp.
Smith, George F.,
[No Organization Registered]
Smith, Vivian A.,
FAA
Snell, Scott,
Computer Sciences Corp.
Spence, Bailey,
Computer Sciences Corp.
Spivey, Cynthia M.,
NASA/GSFC
Squires, Burton E.,
Consultant
Stapko, Ruth,
SATC

Stark, Michael,
NASA/GSFC
Steinberg, Sandee,
Computer Sciences Corp.
Strauss, Dan,
Social Security Administration
Subotic, Anders,
Linkoping University
Sykes, Mari,
Computer Sciences Corp.

Tesoriero, Roseanne,
University of Maryland
Thompson, Sid,
Unisys Corp.
Thornton, Thomas H.,
Jet Propulsion Lab
Travassos, Guilherme H.,
University of Maryland
Trimble, John,
Howard University
Tsagos, Dinos,
DoD

Vane-Harris, Tarik,
Howard University
Vu, Tien-Nhat,
Erricsson Communications

Waligora, Sharon R.,
Computer Sciences Corp.
Wallace, Dolores R.,
NIST
Walsh, Chip,
IRS
Walter, Stephen O.,
Computer Sciences Corp.
Washington, LaVerne B.,
DCMC-Northrop Grumman
Webby, Richard G.,
University of Maryland

Wells, Donna,
Unisys Corp.
Wetzel, Paul E.,
QSI
Whitesell, Steven A.,
Computer Sciences Corp.
Wilson, Robert K.,
Jet Propulsion Lab
Wortman, Kristin,
Computer Sciences Corp.

Wyatt, Valerie S.,
Mountain State
Information Systems, Inc.
Wynne, Denise,
NASD

Yakimovitch, Danil,
University of Maryland
Yin, Meng-Lai,
Raytheon Systems Co.

Youman, Charles,
Blue Cross Blue Shield
NCA

Zelkowitz, Marv,
University of Maryland
Zhang, Zhijun,
University of Maryland

APPENDIX B — STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities. The *Annotated Bibliography of Software Engineering Laboratory Literature* contains an abstract for each document and is available via the SEL Products Page at http://sel.gsfc.nasa.gov/doc-st/docs/bibannot/contents.htm.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, *Proceedings From the First Summer Software Engineering Workshop,* August 1976

SEL-77-002, *Proceedings From the Second Summer Software Engineering Workshop,* September 1977

SEL-78-005, *Proceedings From the Third Summer Software Engineering Workshop,* September 1978

SEL-78-006, *GSFC Software Engineering Research Requirements Analysis Study,* P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, *Applicability of the Rayleigh Curve to the SEL Environment,* T. E. Mapp, December 1978

SEL-78-302, *FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3),* W. J. Decker, W. A. Taylor, et al., July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations,* K. Freburger and V. R. Basili, May 1979

SEL-79-004, *Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment,* C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, *Proceedings From the Fourth Summer Software Engineering Workshop,* November 1979

SEL-80-002, *Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation,* W. J. Decker and C. E. Goorevich, May 1980

SEL-80-005, *A Study of the Musa Reliability Model,* A. M. Miller, November 1980

SEL-80-006, *Proceedings From the Fifth Annual Software Engineering Workshop,* November 1980

SEL-80-007, *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems,* J. F. Cook and F. E. McGarry, December 1980

SEL-80-008, *Tutorial on Models and Metrics for Software Management and Engineering*, V. R. Basili, 1980

SEL-81-011, *Evaluating Software Development by Analysis of Change Data*, D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems*, G. O. Picasso, December 1981

SEL-81-013, *Proceedings of the Sixth Annual Software Engineering Workshop*, December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL)*, A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, *Guide to Data Collection*, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-110, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-305, *Recommended Approach to Software Development*, L. Landis, S. Waligora, F. E. McGarry, et al., June 1992

SEL-81-305SP1, *Ada Developers' Supplement to the Recommended Approach*, R. Kester and L. Landis, November 1993

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982

SEL-82-007, *Proceedings of the Seventh Annual Software Engineering Workshop*, December 1982

SEL-82-008, *Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory*, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, *FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1)*, W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, *Glossary of Software Engineering Laboratory Terms*, T. A. Babst, M. G. Rohleder, and F. E. McGarry, October 1983

SEL-82-1306, *Annotated Bibliography of Software Engineering Laboratory Literature*, D. Kistler, J. Bristow, and D. Smith, November 1994

SEL-83-001, *An Approach to Software Cost Estimation*, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, *Measures and Metrics for Software Development*, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983

SEL-83-007, *Proceedings of the Eighth Annual Software Engineering Workshop*, November 1983

SEL-83-106, *Monitoring Software Development Through Dynamic Variables (Revision 1)*, C. W. Doerflinger, November 1989

SEL-84-003, *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, *Proceedings of the Ninth Annual Software Engineering Workshop*, November 1984

SEL-84-101, *Manager's Handbook for Software Development (Revision 1)*, L. Landis, F. E. McGarry, S. Waligora, et al., November 1990

SEL-85-001, *A Comparison of Software Verification Techniques*, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985

SEL-85-004, *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics*, R. W. Selby, Jr. and V. R. Basili, May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, E. Edwards, F. McGarry, and C. Antle, December 1985

SEL-85-006, *Proceedings of the Tenth Annual Software Engineering Workshop*, December 1985

SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

SEL-86-003, *Flight Dynamics System Software Development Environment (FDS/SDE) Tutorial*, J. Buell and P. Myers, July 1986

SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986

SEL-86-005, *Measuring Software Design*, D. N. Card et al., November 1986

SEL-86-006, *Proceedings of the Eleventh Annual Software Engineering Workshop*, December 1986

SEL-87-001, *Product Assurance Policies and Procedures for Flight Dynamics Software Development*, S. Perry et al., March 1987

SEL-87-002, *Ada® Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

SEL-87-003, *Guidelines for Applying the Composite Specification Model (CSM)*, W. W. Agresti, June 1987

SEL-87-004, *Assessing the Ada® Design Process and Its Implications: A Case Study*, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987

SEL-87-010, *Proceedings of the Twelfth Annual Software Engineering Workshop*, December 1987

SEL-88-001, *System Testing of a Production Ada Project: The GRODY Study*, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988

SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby and L. Esker, December 1988

SEL-88-004, *Proceedings of the Thirteenth Annual Software Engineering Workshop*, November 1988

SEL-88-005, *Proceedings of the First NASA Ada User's Symposium*, December 1988

SEL-89-002, *Implementation of a Production Ada Project: The GRODY Study*, S. Godfrey and C. Brophy, September 1989

SEL-89-004, *Evolution of Ada Technology in the Flight Dynamics Area: Implementation/ Testing Phase Analysis*, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, *Lessons Learned in the Transition to Ada From FORTRAN at NASA/Goddard*, C. Brophy, November 1989

SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989

SEL-89-007, *Proceedings of the Fourteenth Annual Software Engineering Workshop*, November 1989

SEL-89-008, *Proceedings of the Second NASA Ada Users' Symposium*, November 1989

SEL-89-103, *Software Management Environment (SME) Concepts and Architecture (Revision 1)*, R. Hendrick, D. Kistler, and J. Valett, September 1992

SEL-89-301, *Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 3)*, L. Morusiewicz, February 1995

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M. Buhler, K. Pumphrey, and D. Spiegel, March 1990

SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. O. Jun and S. R. Valett, June 1990

SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott and M. Stark, September 1990

SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990

SEL-90-006, *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November 1990

SEL-91-001, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, W. Decker, R. Hendrick, and J. Valett, February 1991

SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report*, E. W. Booth and M. E. Stark, July 1991

SEL-91-004, *Software Engineering Laboratory (SEL) Cleanroom Process Model*, S. Green, November 1991

SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991

SEL-91-006, *Proceedings of the Sixteenth Annual Software Engineering Workshop*, December 1991

SEL-91-102, *Software Engineering Laboratory (SEL) Data and Information Policy (Revision 1)*, F. McGarry, August 1991

SEL-92-001, *Software Management Environment (SME) Installation Guide*, D. Kistler and K. Jeletic, January 1992

SEL-92-002, *Data Collection Procedures for the Software Engineering Laboratory (SEL) Database*, G. Heller, J. Valett, and M. Wild, March 1992

SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992

SEL-92-004, *Proceedings of the Seventeenth Annual Software Engineering Workshop*, December 1992

SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993

SEL-93-002, *Cost and Schedule Estimation Study Report*, S. Condon, M. Regardie, M. Stark, et al., November 1993

SEL-93-003, *Proceedings of the Eighteenth Annual Software Engineering Workshop*, December 1993

SEL-94-001, *Software Management Environment (SME) Components and Algorithms*, R. Hendrick, D. Kistler, and J. Valett, February 1994

SEL-94-003, *C Style Guide*, J. Doland and J. Valett, August 1994

SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994

SEL-94-005, *An Overview of the Software Engineering Laboratory*, F. McGarry, G. Page, V. R. Basili, et al., December 1994

SEL-94-006, *Proceedings of the Nineteenth Annual Software Engineering Workshop*, December 1994

SEL-94-102, *Software Measurement Guidebook (Revision 1)*, M. Bassman, F. McGarry, R. Pajerski, June 1995

SEL-95-001, *Impact of Ada in the Flight Dynamics Division at Goddard Space Flight Center*, S. Waligora, J. Bailey, M. Stark, March 1995

SEL-95-003, *Collected Software Engineering Papers: Volume XIII*, November 1995

SEL-95-004, *Proceedings of the Twentieth Annual Software Engineering Workshop*, December 1995

SEL-95-102, *Software Process Improvement Guidebook (Revision 1)*, K. Jeletic, R. Pajerski,

C. Brown, March 1996

SEL-96-001, *Collected Software Engineering Papers: Volume XIV*, October 1996

SEL-97-001, Guide To Software Engineering Laboratory Data Collection And Reporting, September 1997

SEL-97-002, *Collected Software Engineering Papers: Volume XV*, October 1997

SEL-98-001, *SEL COTS Study Phase 1 - Initial Characterization Study Report*, A. Parra, August 1998

SEL-99-001, *Collected Software Engineering Papers: Volume XVI*, February 1999

SEL-RELATED LITERATURE

[10]Abd-El-Hafiz, S. K., V. R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components", *Proceedings of the IEEE Conference on Software Maintenance-1991 (CSM 91)*, October 1991

[15]Abd-El-Hafiz, S. K., V. R. Basili, "A Knowledge-Based Approach to the Analysis of Loops", *IEEE Transactions on Software Engineering*, May 1996

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study", *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology", *Program Transformation and Programming Environments*. New York: Springer-Verlag, 1984

[1]Bailey, J. W. and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures", *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[8]Bailey, J. W. and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability", *Proceedings of the Eighth Annual National Conference on Ada Technology*, March 1990

[10]Bailey, J. W. and V. R. Basili, "The Software-Cycle Model for Re-Engineering and Reuse", *Proceedings of the ACM Tri-Ada 91 Conference*, October 1991

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering", *ASME Advances in Computer Technology*, January 1980, vol. 1

Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering.* New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology", *Proceedings of the First Pan-Pacific Computer Conference*, September 1985

[7]Basili, V. R., *Maintenance = Reuse-Oriented Software Development*, University of Maryland, Technical Report TR-2244, May 1989

[7]Basili, V. R., *Software Development: A Paradigm for the Future*, University of Maryland, Technical Report TR-2263, June 1989

[8]Basili, V. R., "Viewing Maintenance of Reuse-Oriented Software Development", *IEEE Software*, January 1990

[13]Basili, V. R., "The Experience Factory and Its Relationship to Other Quality Approaches", *Advances in Computers*, vol. 41, Academic Press, Incorporated, 1995

[14]Basili, V. R., "Evolving and Packaging Reading Technologies", P*roceedings of the Third International Conference on Achieving Quality in Software, Florence, Italy*, January 1996

[14]Basili, V. R., "The Role of Experimentation in Software Engineering: Past, Current, and Future", P*roceedings of the Eighteenth Annual Conference on Software Engineering (ICSE-18)*, March 1996

[16]Basili, V. R., "Evolving and Packaging Reading Technologies", *Journal of Systems and Software*, 1997, 38: 3 - 12

[1]Basili, V. R. and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?", *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[13]Basili, V. R., L. Briand, and W. L. Melo, *A Validation of Object-Oriented Design Metrics*, University of Maryland, Computer Science Technical Report, CS-TR-3443, UMIACS-TR-95-40, April 1995

[15]Basili, V. R., L. C. Briand and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", *IEEE Transactions on Software Engineering*, October 1996

[15]Basili, V. R., L. C. Briand, and W. L. Melo, "How Reuse Influences Productivity in Object-Oriented Systems, *Communications of the ACM*, October 1996

[14]Basili, V. R., G. Calavaro, G. Iazeolla, "Simulation Modeling of Software Development Processes", *7th European Simulation Symposium (ESS '95)*, October 1995

[13]Basili, V. R. and G. Caldiera, *The Experience Factory Strategy and Practice*, University of Maryland, Computer Science Technical Report, CS-TR-3483, UMIACS-TR-95-67, May 1995

[9]Basili, V. R., G. Caldiera, and G. Cantone, "A Reference Architecture for the Component Factory", *ACM Transactions on Software Engineering and Methodology*, January 1992

[10]Basili, V. R., G. Caldiera, F. McGarry, et al., "The Software Engineering Laboratory—An Operational Software Experience Factory", *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE 92)*, May 1992

[15]Basili, V. R., S. E. Condon, K. El Emam, R. B. Hendrick and W. L. Melo, "Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components", International Conference on Software Engineering (ICSE-19), May 1997

[1]Basili, V. R. and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory", *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[12]Basili, V. R. and S. Green, "Software Process Evolution at the SEL", *IEEE Software*, July 1994, pp. 58 - 66

[14]Basili, V. R., S. Green, O. Laitenberger, F. Shull, S. Sorumgard, and M. Zelkowitz, "*The Empirical Investigation of Perspective-Based Reading*", University of Maryland, Computer Science Technical Report, CS-TR-3585, UMIACS-TR-95-127, December 1995

[3]Basili, V. R. and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL", *Proceedings of the International Computer Software and Applications Conference*, October 1985

[4]Basili, V. R. and D. Patnaik, *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R. and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation", *Communications of the ACM*, January 1984, vol. 27, no. 1

[1]Basili, V. R. and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory", *Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March 1981

[3]Basili, V. R. and C. L. Ramsey, "ARROWSMITH-P—A Prototype Expert System for Software Engineering Management", *Proceedings of the IEEE/MITRE Expert Systems in Government Symposium*, October 1985

Basili, V. R. and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

Basili, V. R. and R. Reiter, "Evaluating Automatable Measures for Software Development", *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society Press, 1979

[5]Basili, V. R. and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments", *Proceedings of the 9th International Conference on Software Engineering*, March 1987

[5]Basili, V. R. and H. D. Rombach, "TAME: Tailoring an Ada Measurement Environment", *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V. R. and H. D. Rombach, "TAME: Integrating Measurement Into Software Environments", University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R. and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments", *IEEE Transactions on Software Engineering*, June 1988

[7]Basili, V. R. and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment*, University of Maryland, Technical Report TR-2158, December 1988

[8]Basili, V. R. and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*, University of Maryland, Technical Report TR-2446, April 1990

[9]Basili, V. R. and H. D. Rombach, "Support for Comprehensive Reuse", *Software Engineering Journal*, September 1991

[3]Basili, V. R. and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set", *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

Basili, V. R. and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies", *IEEE Transactions on Software Engineering*, December 1987

[3]Basili, V. R. and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology", *Proceedings of the NATO Advanced Study Institute*, August 1985

[5]Basili, V. R. and R. Selby, "Comparing the Effectiveness of Software Testing Strategies", *IEEE Transactions on Software Engineering*, December 1987

[9]Basili, V. R. and R. W. Selby, "Paradigms for Experimentation and Empirical Studies in Software Engineering", *Reliability Engineering and System Safety*, January 1991

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering", *IEEE Transactions on Software Engineering*, July 1986

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects", *IEEE Transactions on Software Engineering*, November 1983

[2]Basili, V. R. and D. M. Weiss, *A Methodology for Collecting Valid Software Engineering Data*, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R. and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data", *IEEE Transactions on Software Engineering*, November 1984

[1]Basili, V. R. and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives", *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*, August 1977

Basili, V. R. and M. V. Zelkowitz, "Designing a Software Measurement Experiment", *Proceedings of the Software Life Cycle Management Workshop*, September 1977

[1]Basili, V. R. and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory", *Proceedings of the Second Software Life Cycle Management Workshop*, August 1978

[1]Basili, V. R. and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment", *Computers and Structures*, August 1978, vol. 10

Basili, V. R. and M. V. Zelkowitz, "Analyzing Medium Scale Software Development", *Proceedings of the Third International Conference on Software Engineering.* New York: IEEE Computer Society Press, 1978

[13]Basili, V. R., M. Zelkowitz, F. McGarry, G. Page, S. Waligora, and R. Pajerski, "SEL's Software Process-Improvement Program", *IEEE Software*, vol. 12, no. 6, November 1995, pp. 83 - 87

[12]Bassman, M. J., F. McGarry, and R. Pajerski, *Software Measurement Guidebook*, NASA-GB-001-94, Software Engineering Program, July 1994

[9]Booth, E. W. and M. E. Stark, "Designing Configurable Software: COMPASS Implementation Concepts", *Proceedings of Tri-Ada 1991*, October 1991

[10]Booth, E. W. and M. E. Stark, "Software Engineering Laboratory Ada Performance Study—Results and Implications", *Proceedings of the Fourth Annual NASA Ada User's Symposium*, April 1992

[10]Briand, L. C. and V. R. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process", *Proceedings of the 1992 IEEE Conference on Software Maintenance (CSM 92)*, November 1992

[10]Briand, L. C., V. R. Basili, and C. J. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development", *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

[11]Briand, L. C., V. R. Basili, and C. J. Hetmanski, *Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components*, University of Maryland, Technical Report TR-3048, March 1993

[12]Briand, L. C., V. R. Basili, Y. Kim, and D. R. Squier, "A Change Analysis Process to Characterize Software Maintenance Projects", *Proceedings of the International Conference on Software Maintenance*, Victoria, British Columbia, Canada, September 19 - 23, 1994, pp. 38 - 49

[9]Briand, L. C., V. R. Basili, and W. M. Thomas, *A Pattern Recognition Approach for Software Engineering Data Analysis*, University of Maryland, Technical Report TR-2672, May 1991

[14]Briand, L., V. R. Basili, S. Condon, Y. Kim, W. Melo and J. D. Valett, "Understanding and Predicting the Process of Software Maintenance Releases", *Proceedings of the Eighteenth Annual Conference on Software Engineering (ICSE-18)*, March 1996

[14]Briand, L., Y. Kim, W. Melo, C. B. Seaman, V. R. Basili, "Qualitative Analysis for Maintenance Process Assessment", University of Maryland, Computer Science Technical Report, CS-TR-3592, UMIACS-TR-96-7, January 1996

[16]Briand, L., Y. Kim, W. Melo, C. Seaman, and V. R. Basili, Q-MOPP: Qualitative Evaluation of Maintenance Organizations, Processes and Products, *Journal of Software Maintenance: Research and Practice*, 10, 249 - 278 1998

[13]Briand, L., W. Melo, C. Seaman, and V. R. Basili, "Characterizing and Assessing a Large-Scale Software Maintenance Organization", *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, U.S.A., April 23 - 30, 1995

[11]Briand, L. C., S. Morasca, and V. R. Basili, "Measuring and Assessing Maintainability at the End of High Level Design", *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM 93)*, November 1993

[12]Briand, L., S. Morasca, and V. R. Basili, *Defining and Validating High-Level Design Metrics*, University of Maryland, Computer Science Technical Report, CS-TR-3301, UMIACS-TR-94-75, June 1994

[13]Briand, L., S. Morasca, and V. R. Basili, *Property-Based Software Engineering Measurement*, University of Maryland, Computer Science Technical Report, CS-TR-3368, UMIACS-TR-94-119, November 1994

[13]Briand, L., S. Morasca, and V. R. Basili, *Goal-Driven Definition of Product Metrics Based on Properties*, University of Maryland, Computer Science Technical Report, CS-TR-3346, UMIACS-TR-94-106, December 1994

[15]Briand, L., S. Morasca, and V. R. Basili, "Property-Based Software Engineering Measurement, *IEEE Transactions on Software Engineering*, January 1996

[15]Briand, L., S. Morasca, and V. R. Basili, Response to: Comments on "Property-Based Software Engineering Measurement: Refining the Additivity Properties", *IEEE Transactions on Software Engineering*, March 1997

[11]Briand, L. C., W. M. Thomas, and C. J. Hetmanski, "Modeling and Managing Risk Early in Software Development", *Proceedings of the Fifteenth International Conference on Software Engineering (ICSE 93)*, May 1993

[5]Brophy, C. E., W. W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods", *Proceedings of the Joint Ada Conference*, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project", *Proceedings of the Washington Ada Technical Conference*, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size", Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation", Computer Sciences Corporation, Technical Memorandum, November 1982

[3]Card, D. N., "A Software Technology Evaluation Program", *Annais do XVIII Congresso Nacional de Informatica*, October 1985

[5]Card, D. N. and W. W. Agresti, "Resolving the Software Science Anomaly", *Journal of Systems and Software*, 1987

[6]Card, D. N. and W. W. Agresti, "Measuring Software Design Complexity", *Journal of Systems and Software*, June 1988

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices", *IEEE Transactions on Software Engineering*, February 1986

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System", Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules", Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D. N., F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies", *IEEE Transactions on Software Engineering*, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization", *Proceedings of the Eighth International Conference on Software Engineering.* New York: IEEE Computer Society Press, 1985

[1]Chen, E. and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies", *Proceedings of the Fifth International Conference on Software Engineering.* New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes", *ACM Software Engineering Notes*, July 1986

[15]Condon, S., R. Hendrick, M. E. Stark, W. Steger, "The Generalized Support Software (GSS) Domain Engineering Process: An Object-Oriented Implementation and Reuse Success at Goddard Space Flight Center", *Addendum to the Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 96)*, San Jose, California, U.S.A., October 1996

[15]Devanbu, P., S. Karstu, W. L. Melo and W. Thomas, "Analytical and Empirical Evaluation of Software Reuse Metrics, *Proceedings of the 18th International Conference on Software Engineering (ICSE-18),* March 1996

[2]Doerflinger, C. W. and V. R. Basili, "Monitoring Software Development Through Dynamic Variables", *Proceedings of the Seventh International Computer Software and Applications Conference*. New York: IEEE Computer Society Press, 1983

Doubleday, D., *ASAP: An Ada Static Source Code Analyzer Program*, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S. and C. Brophy, "Experiences in the Implementation of a Large Ada Project", *Proceedings of the 1988 Washington Ada Symposium*, June 1988

[5]Jeffery, D. R. and V. R. Basili, *Characterizing Resource Data: A Model for Logical Association of Software Data*, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R. and V. R. Basili, "Validating the TAME Resource Data Model", *Proceedings of the Tenth International Conference on Software Engineering*, April 1988

[16]Kontio, J., *The Riskit Method for Software Risk Management, Version 1.00*, University of Maryland, Computer Science Technical Report, CS-TR-3782, UMIACS-TR-97-38, (Date)

[16]Kontio, J. and V. R. Basili, Empirical Evaluation of a Risk Management Method, *SEI Conference on Risk Management*, 1997

[15]Kontio, J., G. Caldiera, and V. R. Basili, "Defining Factors, Goals and Criteria for Reusable Component Evaluation", *CASCON '96 Conference*, November 1996

[15]Kontio, J., H. Englund, and V. R. Basili, *Experiences from an Exploratory Case Study with a Software Risk Management Method*, Computer Science Technical Report, CS-TR-3705, UMIACS-TR-96-75, August 1996

[15]Lanubile, F., "Why Software Reliability Predictions Fail", *IEEE Software*, July 1996

[16]Lanubile, F., "Empirical Evaluation of Software Maintenance Technologies", Empirical Software Engineering, 2, 97-108, 1997

[11]Li, N. R. and M. V. Zelkowitz, "An Information Model for Use in Software Management Estimation and Prediction", *Proceedings of the Second International Conference on Information Knowledge Management*, November 1993

[5]Mark, L. and H. D. Rombach, *A Meta Information Base for Software Engineering*, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L. and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications", *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[5]McGarry, F. E. and W. W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)", *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment", *Proceedings of the Sixth Washington Ada Symposium (WADAS)*, June 1989

[13]McGarry, F., R. Pajerski, G. Page, et al., *Software Process Improvement in the NASA Software Engineering Laboratory*, Carnegie-Mellon University, Software Engineering Institute, Technical Report CMU/SEI-94-TR-22, ESC-TR-94-022, December 1994

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product", *Proceedings of the Hawaiian International Conference on System Sciences*, January 1985

[15]Morasca, S., L. C. Briand, V. R. Basili, E. J. Weyuker and M. V. Zelkowitz, Comments on "Towards a Framework for Software Measurement Validation", *IEEE Transactions on Software Engineering*, March 1997

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation", *Proceedings of the Eighth International Computer Software and Applications Conference*, November 1984

[12]Porter, A. A., L. G. Votta, Jr., and V. R. Basili, *Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment*, University of Maryland, Technical Report TR-3327, July 1994

[5]Ramsey, C. L. and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management", *IEEE Transactions on Software Engineering*, June 1989

[3]Ramsey, J. and V. R. Basili, "Analyzing the Test Process Using Structural Coverage", *Proceedings of the Eighth International Conference on Software Engineering.* New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability", *IEEE Transactions on Software Engineering*, March 1987

[8]Rombach, H. D., "Design Measurement: Some Lessons Learned", *IEEE Software*, March 1990

[9]Rombach, H. D., "Software Reuse: A Key to the Maintenance Problem", *Butterworth Journal of Information and Software Technology*, January/February 1991

[6]Rombach, H. D. and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study", *Proceedings From the Conference on Software Maintenance*, September 1987

[6]Rombach, H. D. and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases", *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[7]Rombach, H. D. and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL*, University of Maryland, Technical Report TR-2252, May 1989

[10]Rombach, H. D., B. T. Ulery, and J. D. Valett, "Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL", *Journal of Systems and Software*, May 1992

[14]Seaman, C. B., V. R. Basili, *"Communication and Organization in Software Development: An Empirical Study"*, University of Maryland, Computer Science Technical Report, CS-TR-3619, UMIACS-TR-96-23, April 1996

[15]Seaman, C. B, V. R. Basili, "An Empirical Study of Communication in Code Inspection", *Proceedings of 19th International Conference on Software Engineering (ICSE-19),* May 1997, pp. 96-106

[16]Seaman, C. B and V. R. Basili, "An Empirical Study of Communication in Code Inspections", University of Maryland,

[16]Seaman, C. B, V. R. Basili, "Communication and Organization in Software Development: An Empirical Study", *IBM Systems Journal*, Vol. 36, No. 4, 1997

[16]Seaman, C. B, V. R. Basili, The Study of Software Maintenance Organizations and Processes

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada", *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience", *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach", *Proceedings of the CASE Technology Conference*, April 1988

[9]Seidewitz, E., "Object-Oriented Programming Through Type Extension in Ada 9X", *Ada Letters*, March/April 1991

[10]Seidewitz, E., "Object-Oriented Programming With Mixins in Ada", *Ada Letters*, March/April 1992

[12]Seidewitz, E., "Genericity versus Inheritance Reconsidered: Self-Reference Using Generics", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1994

[4]Seidewitz, E. and M. Stark, "Towards a General Object-Oriented Software Development Methodology", *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[9]Seidewitz, E. and M. Stark, "An Object-Oriented Approach to Parameterized Software in Ada", *Proceedings of the Eighth Washington Ada Symposium*, June 1991

[8]Stark, M., "On Designing Parametrized Systems Using Ada", *Proceedings of the Seventh Washington Ada Symposium*, June 1990

[11]Stark, M., "Impacts of Object-Oriented Technologies: Seven Years of SEL Studies", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1993*

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse", *Proceedings of TRI-Ada 1989*, October 1989

[5]Stark, M. and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle", *Proceedings of the Joint Ada Conference*, March 1987

[15]Stark, M., "Using Applet Magic (tm) to Implement an Orbit Propagator: New Life for Ada Objects", *Proceedings of the 14th Annual Washington Ada Symposium (WAdaS97)*, June 1997

[13]Stark, M. and E. Seidewitz, "Generalized Support Software: Domain Analysis and Implementation", *Addendum to the Proceedings OOPSLA '94*, Ninth Annual Conference, Portland, Oregon, U.S.A., October 1994, pp. 8 - 13

[10]Straub, P. A. and M. V. Zelkowitz, "On the Nature of Bias and Defects in the Software Specification Process", *Proceedings of the Sixteenth International Computer Software and Applications Conference (COMPSAC 92)*, September 1992

[8]Straub, P. A. and M. V. Zelkowitz, "PUC: A Functional Specification Language for Ada", *Proceedings of the Tenth International Conference of the Chilean Computer Science Society*, July 1990

[7]Sunazuka, T. and V. R. Basili, *Integrating Automated Support for a Software Management Cycle Into the TAME System*, University of Maryland, Technical Report TR-2289, July 1989

[13]Thomas, W. M., A. Delis, and V. R. Basili, *An Analysis of Errors in a Reuse-Oriented Development Environment*, University of Maryland, Computer Science Technical Report, CS-TR-3424, UMIACS-TR-95-24

[16]Tesoriero, R., M. Zelkowitz, A Model of Noisy Software Engineering Data (Status Report), *Proceedings of the Twentieth International Conference on Software Engineering*, April 19 - 25, 1998

[16]Tesoriero, R., M. Zelkowitz, WEBME: A Web-based tool For Data Analysis and Presentation, IEEE Internet Computing

[16]Thomas, W. M., A. Delis, and V. R. Basili An Analysis of Errors in a Reuse-Oriented Development Environment, *Journal of Systems Software*, 38:211 - 224, 1997

[10]Tian, J., A. Porter and M. V. Zelkowitz, "An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity", *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

Turner, C. and G. Caron, *A Comparison of RADC and NASA/SEL Software Development Data*, Data and Analysis Center for Software, Special Publication, May 1981

[10]Valett, J. D., "Automated Support for Experience-Based Software Management", *Proceedings of the Second Irvine Software Symposium (ISS_92)*, March 1992

[5]Valett, J. D. and F. E. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory", *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[14]Waligora, S., J. Bailey, and Mike Stark, "The Impact of Ada and Object-Oriented Design in NASA Goddard's Flight Dynamics Division", July 1996

[3]Weiss, D. M. and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory", *IEEE Transactions on Software Engineering*, February 1985

[5]Wu, L., V. R. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems", *Proceedings of the Joint Ada Conference*, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium-Scale Software Projects", *Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science.* New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research", *Empirical Foundations for Computer and Information Science* (Proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study", *Proceedings of the 26th Annual Technical Symposium of the Washington, D.C., Chapter of the ACM*, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development", *Journal of Systems and Software*, 1988

[8]Zelkowitz, M. V., "Evolution Towards Specifications Environment: Experiences With Syntax Editors", *Information and Software Technology*, April 1990

[14]Zelkowitz, M. V., "Software Engineering Technology Infusion Within NASA", *IEEE Transactions On Engineering Management*, vol. 43, no. 3, August 1996

NOTES:

[1]This article also appears in SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982.

[2]This article also appears in SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983.

[3]This article also appears in SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985.

[4]This article also appears in SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986.

[5]This article also appears in SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987.

[6]This article also appears in SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988.

[7]This article also appears in SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989.

[8]This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.

[9]This article also appears in SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991.

[10]This article also appears in SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992.

[11]This article also appears in SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993.

[12]This article also appears in SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994.

[13]This article also appears in SEL-95-003, *Collected Software Engineering Papers: Volume XIII*, November 1995.

[14]This article also appears in SEL-96-001, *Collected Software Engineering Papers: Volume XIV*, October 1996.

[15]This article also appears in SEL-97-002, *Collected Software Engineering Papers: Volume XV*, October 1997.

[16]This article also appears in SEL-99-001, *Collected Software Engineering Papers: Volume XVI*, March 1999.